[19]中华人民共和国国家知识产权局

[51]Int. Cl[7]
G06F 9/48
H04L 9/00

[12]发明专利申请公开说明书

[21]申请号 01103000.3

[43]公开日 2001年8月22日

[11]公开号 CN 1309351A

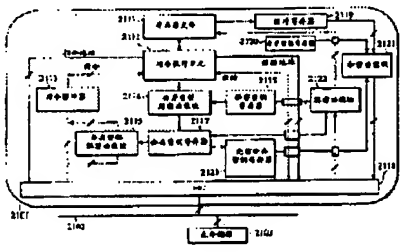
[22]申请日 2001.2.14 [21]申请号 01103000.3
[30]优先权
[32]2000.2.14 [33]JP [31]035898/2000
[32]2000.5.8 [33]JP [31]135010/2000
[71]申请人 株式会社东芝
地址 日本神奈川县
[72]发明人 桥本干生 寺本圭一 齐藤健
白川健治 藤本谦作
[74]专利代理机构 中国国际贸易促进委员会专利商标事务所
代理人 吴丽丽

权利要求书4页 说明书43页 附图页数15页

[54]发明名称 抗干预微处理器

[57]摘要

在多任务环境下，抗干预微处理器保存一个其执行被中断的程序 的上下文信息，其中该上下文信息含有指明该程序的执行状态和该程 序的执行码密钥的信息。通过从保存的上下文信息恢复该程序的执行 状态，可以重新启动该程序的执行。利用微处理器的公开密钥可以将 此上下文信息加密，然后利用微处理器的秘密密钥进行解密。



ISSN1008-4274

01.02.14

权 利 要 求 书

1. 一种具有不能被读出到外部的唯一秘密密钥和与该唯一秘密密钥对应的唯一公开密钥的微处理器，该微处理器包括：

读取单元，被进行配置以从外部存储器读出多个利用不同执行码密钥加密的程序；

解密单元，被进行配置以利用各自解密密钥，对多个通过读取单元读出的程序进行解密；

执行单元，被进行配置以执行多个利用解密单元解密的程序；

上下文信息保存单元，被进行配置以将其执行被中断的一个程序的上下文信息保存到外部存储器或保存到在微处理器内部设置的上下文信息存储器，该上下文信息含有指明此程序的执行状态和此程序的执行码密钥的信息；以及

重新启动单元，被进行配置以通过从外部存储器或上下文信息存储器读出上下文信息并通过从上下文信息中恢复此程序的执行状态，重新启动执行此程序。

2. 根据权利要求 1 所述的微处理器，其中所配置的上下文信息保存单元利用公开密钥对上下文信息进行加密，并将加密上下文信息保存到外部存储器；以及

所配置的重新启动单元通过从外部存储器读出加密上下文信息，利用秘密密钥解密加密上下文信息，以及从解密上下文信息中恢复一个程序的执行状态，重新启动此程序的执行。

3. 根据权利要求 2 所述的微处理器，其中仅当包含在解密上下文信息内的解密执行码密钥与此程序的执行码密钥一致时，重新启动单元才重新启动此程序的执行.

4. 根据权利要求 2 所述的微处理器，其中重新启动单元将包含在解密上下文信息内的解密执行码密钥用作解密密钥以解密此程序.

5. 根据权利要求 1 所述的微处理器，其中所配置的上下文信息保存单元以明文形式将上下文信息保存到此程序被中断后所执行的另一

个程序不可读的上下文信息存储器；以及

通过从上下文信息存储器读出上下文信息并从上下文信息恢复此程序的执行码，所配置的重新启动单元重新启动此程序的执行。

6. 根据权利要求 5 所述的微处理器，其中重新启动单元根据另一个程序规定的指令重新启动此程序的执行。

7. 根据权利要求 6 所述的微处理器，其中在此程序的执行被中断时，上下文信息保存单元将上下文信息保存到上下文信息存储器，并利用公开密钥将上下文信息存储器内的上下文信息加密，然后根据另一个程序规定的另一条指令的执行，将加密上下文信息存储到外部存储器。

8. 根据权利要求 5 所述的微处理器，其中在此程序的执行被中断时，上下文信息保存单元将上下文信息保存到上下文信息存储器，利用公开密钥将上下文信息存储器内的上下文信息加密，然后将加密上下文信息存储到另一个程序规定的外部存储器。

9. 根据权利要求 1 所述的微处理器，其中所配置的上下文信息保存单元产生作为临时密钥的随机数、加密上下文信息、然后将加密上下文信息存储到外部存储器，加密上下文信息含有：第一数值，通过对信息进行加密获得，利用临时密钥指明此程序的执行状态；以及第二数值，通过利用公开密钥加密临时密钥获得；以及

通过从外部存储器读出加密上下文信息，利用秘密密钥由包含在加密上下文信息内的第二数值解密获得临时密钥，利用解密的临时密钥由包含在加密上下文信息内第一数值解密出指明执行状态的信息，以及从解密上下文信息恢复此程序的执行状态，所配置的重新启动单元重新启动此程序的执行。

10. 根据权利要求 9 所述的微处理器，其中上下文信息保存单元保存还含有利用此程序的执行码密钥对临时密钥进行加密获得的第三数值的加密上下文信息。

11. 根据权利要求 10 所述的微处理器，其中重新启动单元利用秘密密钥由包含在加密上下文信息内的第二数值解密获得第一临时密

钥，并利用第一解密临时密钥由包含在加密上下文信息内的第一数值解密获得指明执行状态的信息，同时利用该程序的执行码密钥由包含在加密上下文信息内的第三数值解密获得第二临时密钥，然后只在第一解密的临时密钥与第二解密的临时密钥一致时，重新启动此程序的执行。

12. 根据权利要求 1 所述的微处理器，该微处理器进一步包括：

执行状态存储单元，用于存储当前执行程序的执行状态；以及

执行状态初始化单元，被进行配置以在此程序被中断后而在另一个程序开始之前，将执行状态存储单元的内容初始化为规定数值或将执行状态存储单元的内容加密。

13. 根据权利要求 1 所述的微处理器，该微处理器进一步包括：

密钥读取单元，被进行配置以从外部存储器读出被事先利用公开密钥加密的各程序的执行码密钥；以及

密钥解密单元，被进行配置以利用秘密密钥解密通过密钥读取单元读出的执行码密钥；

其中解密单元利用作为解密密钥的执行码密钥解密各程序。

14. 根据权利要求 1 所述的微处理器，该微处理器进一步包括：

执行状态存储单元，用于存储当前执行程序的执行状态和将被当前执行程序处理的数据的加密属性；以及

数据加密单元，被进行配置以根据存储在执行状态存储单元的加密属性对将由当前执行程序处理的数据进行加密。

15. 根据权利要求 1 所述的微处理器，该微处理器进一步包括：

执行状态存储单元，用于存储当前执行程序的执行状态、将被当前执行程序处理的数据的加密属性以及用于规定加密属性的加密属性规定信息；

相关信息写入单元，被进行配置以将涉及加密属性规定信息并含有利用秘密密钥获得的签名的相关信息写入外部存储器；

相关信息读出单元，被进行配置以根据将由当前执行程序引用的数据的地址从外部存储器读出相关信息；

数据引用许可单元，被进行配置以利用公开密钥验证包含在相关信息内的签名，并且只有当相关信息内的签名与微处理器的原始签名一致时，才允许当前执行程序根据相关信息和规定信息的加密属性，通过确定密钥和用于数据引用的算法进行数据引用；以及

数据加密单元，被进行配置以根据存储在执行状态存储单元的加密属性将由当前执行程序引用的数据加密。

16. 根据权利要求 1 所述的微处理器，该微处理器进一步包括：

高速缓冲存储器，用于以高速缓存行为单位高速缓存多个程序的明文指令和明文数据，该高速缓冲存储器具有属性区用于各高速缓存行，指明在解密其指令被高速缓存到各高速缓存行的各程序或其执行会将明文数据高速缓存到各高速缓存行的各程序时用于唯一标识解密密钥的解密密钥标识符；

高速缓存访问控制单元，被进行配置以只有当加密属性对一个高速缓存行指明的解密密钥标识符与加密属性对另一个高速缓存行指明的解密密钥标识符一致时，允许通过根据另一个高速缓存行内的一个高速缓存数据执行一个存储在一个高速缓存行的高速缓存程序引起的数据引用。

17. 根据权利要求 16 所述的微处理器，其中当不允许进行数据引用时，将新数据从外部存储器高速缓存到另一个高速缓存行。

18. 根据权利要求 16 所述的微处理器，其中当不允许进行数据引用时，保护异常中断此高速缓存程序的执行。

19. 根据权利要求 1 所述的微处理器，其中执行单元还执行明文程序，并具有调试功能块，在明文程序的执行期间，当执行特定地址或地址区域的程序时或将数据引用到特定地址或地址区域的数据时，该调试功能块用于产生异常，在执行加密程序期间，此调试程序无效。

20. 根据权利要求 1 所述的微处理器，其中该微处理器的各组成单元包含在单一芯片或单一封装内。

01.02.14

说　　明　　书

抗干预微处理器

本发明涉及可以在多任务程序执行环境下防止非法变更执行码和非法处理目标数据的微处理器。

最近几年，微处理器的性能得到显著改善，以致微处理器除了具有传统的诸如计算和图形功能外，还可以实现对视频图像和音频声音的再生和编辑。通过在为最终用户设计的系统（以下简称：PC）中实现这种微处理器，用户可以在监视器上欣赏各种视频图像和音频声音。此外，通过将 PC 的再生视频图像和音频声音的功能与计算能力相结合，可以改善对游戏等的适用性。这种微处理器不是专为某种特定硬件设计的而是可以在各种硬件中实现，因此其优势在于，通过简单更换执行程序的微处理器，已经拥有 PC 的用户花费不多就可以欣赏视频图像和音频声音的再生和编辑。

如果在 PC 上处理视频图像和音频声音，就会产生原始图像和音乐的版权保护问题。在 MD 或数字视频重放装置中，通过在这些装置中事先实现防止非法复制的机制，可以防止无限复制。虽然这种装置还在制造，但是试图通过拆除或改变这些装置来进行非法复制的情况却很少，而且世界范围内的趋势是通过法律禁止制造和销售为了进行非法复制能够改变的装置。因此，由于基于硬件进行非法复制造成的损害并不很严重。

然而，在 PC 上对图像数据和音乐数据进行处理是通过软件进行的而不是通过硬件进行的，并且最终用户可以在 PC 上随意改变软件。即，如果用户具有某些知识，则通过分析程序并重写可执行软件，可以非常容易地进行非法复制。此外，不同于硬件的问题是，这样产生的用于非法复制的软件可以通过诸如网络的各种媒体迅速传播。

为了解决这些问题，用于再生诸如商业电影或音乐的版权保护内容的 PC 软件，传统上采用一种通过对软件进行加密防止分析和变更

01.02.14

的技术。这种技术就是抗干预软件（参考 David Aucsmith 等人在 Proceedings of the 1996 Intel Software Developer's Conference 上发表的 "Tamper Resistant Software: An Implementation"）。

在防止通过 PC 向用户提供的有价值信息（不仅包括视频数据和音频数据而且包括文本和技术诀窍）的非法复制方面，以及在防止 PC 软件本身的技术诀窍被分析方面，抗干预软件技术仍然有效。

然而，抗干预软件技术是一种，通过在开始执行程序之前对要求保护的程序的一部分进行加密，在执行该部分之前对该部分立即进行解密并在该部分执行完毕后立即对该部分再加密，使得难于利用诸如反汇编程序或调试程序的软件工具进行分析。因此，只要处理器可以执行该程序，通过从程序的启动处开始一步一步进行分析总可以对程序进行分析。

此事实成为版权所有人向系统提供版权保护内容用于利用 PC 再生视频数据和音频数据的障碍。

在这方面，其它抗干预软件应用程序也易受攻击，并且此事实成为通过 PC 进行高级信息服务和将含有企业或个人技术诀窍的程序应用到 PC 的障碍。

总之，在软件保护方面同样存在这些问题，此外，PC 是开放式平台，因此存在通过变更被确定为系统软件配置基础的操作系统（OS）进行攻击问题。换句话说，通过使用属于 OS 的特权，怀有恶意的熟练用户可以变更其自有 PC 的 OS 来废除或分析插入到应用程序内的版权保护机制。

当前的 OS 通过利用对存储器的特权操作功能和 CPU 中提供的特权执行控制功能，在计算机的控制下进行资源管理和资源使用仲裁。管理的目标包括传统目标（例如：设备、CPU 和存储资源）以及网络层或应用层 QoS（服务质量）。尽管如此，资源管理的基础仍然是对执行程序所需的资源进行配置。换句话说，分配 CPU 时间来执行此程序并将分配执行程序所需的存储空间是资源管理的基础。通过控制实现访问这些资源的程序的执行（通过分配 CPU 的时间和存储空间），对

其它设备、网络和应用层服务质量 Qos 进行控制.

OS 具有执行 CPU 时间分配和存储空间分配的特权. 换句话说, 为了对 CPU 分配时间, OS 具有在任意时间中断并重新启动应用程序的特权并具有在任意时间将分配到应用程序的存储空间的内容转移到不同分层的存储空间的特权. (通常) 通过利用应用程序的不同访问速度和访问能力隐匿分层存储系统, 将分配到应用程序的存储空间的内容转移到不同分层的存储空间的特权还用于为应用程序提供平面存储器空间.

使用这两种特权, OS 可以中断应用程序的执行状态并在任意时间对它进行快速转储, 并且在对它进行拷贝或重写之后重新启动它. 此功能还可以被用作分析隐藏在应用程序内的秘密的工具.

为了在计算机上防止应用程序被分析, 有几种对程序或数据进行加密的公知技术(例如: Hampson, 第 4, 847, 902 号美国专利、Hartman, 第 5, 224, 166 号美国专利、Davis, 第 5, 806, 706 号美国专利、Takahashi 等, 第 5, 825, 878 号美国专利、Buer 等人, 第 6, 003, 117 号美国专利、第 11 - 282667 号日本公开专利申请 (1999)). 然而, 这些公知的技术均未涉及防止程序运行过程和数据秘密被 OS 进行上述特权操作问题.

基于 Intel 公司开发的 X86 结构的传统技术 (Hartman, 第 5, 224, 166 号美国专利) 是一种通过利用规定的密钥 Kx 对执行码和数据进行加密以存储执行码和执行数据的技术. 密钥 Kx 可以被表示为 $E_{kp}[Kx]$ 的形式, 利用与嵌入到处理器内的秘密密钥 Ks 对应的公开密钥 Kp, 可以对 $E_{kp}[Kx]$进行加密. 因此, 只有知道 Ks 的处理器可以对存储器上的加密执行码进行解密. 将密钥 Kx 存储到处理器内被称为段式寄存器的寄存器.

利用这种机制, 通过对代码进行加密在某种程度上可以避免用户发现程序代码的秘密. 此外, 对于不知道代码密钥 Kx 的人来说, 由于密码原因难于根据其内涵或利用密钥 Kx 解密时可执行的新产生代码来变更代码.

然而, 采用这种技术的系统的缺点在于, 利用被称为上下文切换

的 OS 特权有可能对程序进行分析，而无需对加密的执行码进行解密。

更具体地说，当利用中断停止执行程序或当预期系统调用程序自行调用软件中断命令时，为了执行其它程序，OS 进行上下文切换。上下文切换操作将指明该点寄存器值的集合的程序执行状态（以下简称为：上下文信息）存储到存储器，并将事先存储到存储器的另一个程序的上下文信息再存入寄存器。

图 15 示出在 x86 处理器中使用的传统上下文存储格式。这里存储了应用程序使用的寄存器的所有内容。当再启动被中断的程序时，将该程序的上下文信息再存入寄存器。为了并行运行多个程序，上下文切换是不可缺少的功能。在传统技术中，在上下文切换时，OS 可以读取寄存器值，因此根据该程序的执行状态是如何改变的，即使不是全部，也可以猜测出该程序执行的大多数操作。

此外，通过控制在此时通过设置计时器等产生异常的时间，在程序的任意执行点可以进行此处理。除了中断执行和分析之外，还可以恶意重写寄存器信息。重写寄存器不仅可以改变程序运行而且可以使得对程序进行分析更容易。OS 可以存储应用程序的任意状态，因此通过重写寄存器值并通过反复运行程序，可以分析程序的运行。除了上述功能之外，处理器还具有诸如逐步执行的调试支持功能，存在的问题是，利用所有这些功能，OS 可以对应用程序进行分析。

就数据而论，第 5, 224,166 号美国专利认为，仅通过利用加密代码段执行程序，程序可以访问加密数据。这里存在的问题是加密程序利用任意密钥可以自由读取加密数据，而与对程序加密的密钥无关，即使存在利用互相不同的密钥加密的程序。这种传统技术中未说明这些情况，即 OS 和应用程序独立具有它们自己的秘密并且应用程序的秘密不被 OS 发现，或者多个程序供应商分别具有它们自己的秘密。

当然，即使是在现有的处理器中，也可以在应用程序之间划分存储空间并利用虚拟存储机制提供的保护功能来禁止应用程序访问系统存储器。然而，只要虚拟存储制受 OS 的控制，则对应用程序秘密的保护就不能依赖于 OS 控制下的功能。这是由于 OS 可以忽略保护机制

访问数据，并且在提供上述虚拟存储器方面，这种特权不可缺少。

作为另一种传统技术，第 11－282667（1999）号日本公开专利申请公开了一种技术，这种技术为了存储应用程序的秘密信息而在 CPU 内设置秘密存储器。在这种技术中，为了访问秘密存储器内的数据，需要规定基准值。但是，此技术未披露如何防止同一个 CPU 内运行的多个程序（特别是 OS）使用用于获得对秘密数据的访问权的基准数值.

此外，在第 5, 123, 045 号美国专利中，Ostrovsky 等人公开了一种系统，该系统的先决条件是使用具有与应用程序对应的唯一秘密密钥的子处理器，在该系统中，不能根据这些子处理器访问主存储器上的程序的访问方式来推测程序运行．这是基于，通过将根据存储器实现运行的指令系统转换到与此指令系统不同的另一个指令系统，实现随机存储访问的机制。

然而，对不同的应用程序，这种技术要求不同的子处理器，因此这种技术的成本高，并且预期用于处理这种指令系统的编译程序和处理器硬件的执行和快速实现过程非常困难，这是由于它们与当前使用的处理器的编译程序和处理器硬件非常不同．除此之外，与上述说明的将程序码和数据简单加密的其它传统技术（例如：第 5, 224,166 号美国专利和第 11－282667 号日本公开专利申请）比较，在这种处理器中，即使当数据和实际操作码的运行被观察到并被跟踪以致调试程序变得非常困难时，难于包含数据内容与运行之间的对应之处，因此，这种技术存在许多实际问题。

因此，本发明的第一个目的是提供一种微处理器，该微处理器即使是在被中断停止执行时也可以防止在多任务环境下内部执行的算法和存储区内的数据状态被非法分析.

此第一个目的受传统技术能够保护程序码的数值而不能防止利用通过发生异常或调试功能中断程序的执行进行分析的启发．因此，本发明的目的是提供一种即使是在程序执行中断时仍能确实保护代码的微处理器，在此微处理器中，这种保护与当前 OS 要求的执行控制功能和存储器管理功能兼容。

01-02-14

本发明的第二个目的是提供一种即使执行多个利用不同密钥加密的程序，其各程序均可以独立获得正确可读/可写数据区的微处理器。

此第二个目的是受第 5, 224,166 号美国专利公开的传统技术的启发，该技术仅提供简单保护，其中禁止利用非加密代码访问加密数据区，并且不可能独立地对多个程序保护它们的秘密。因此，本发明的目的还在于提供一种当多个应用程序具有它们各自的（加密的）秘密时具有用于防止各应用程序的秘密被 OS 使用的数据区的微处理器。

本发明的第三个目的是提供一种可以防止上述数据区的保护属性（即加密属性）被 OS 非法重写的微处理器。

此第三个目的是受第 5, 224, 166 号美国专利公开的传统技术的启发，该技术的缺点在于，通过利用上下文切换中断程序的执行，OS 可以重写在段式寄存器内设置的加密属性。一旦，通过重写加密属性，程序进入以明文形式写入数据的状态，不加密就不将数据写入存储器。即使在某些时间应用程序校验段式寄存器的数值，但是，如果此后重写寄存器的数值，则结果相同。因此，本发明的目的还在于提供一种微处理器，该微处理器具有可以禁止这种变更或可以检测这种变更并可以对这种变更采取适当措施的机制。

本发明的第四个目的是提供一种微处理器，该微处理器可以防止加密属性受密码分析原理的所谓选择明文攻击法攻击，在密码分析原理中，程序可以使用任意数值作为数据密钥。

本发明的第五个目的是提供一种微处理器，该微处理器具有程序调试和反馈的机制。换句话说，本发明目的在于提供一种处理器，在该微处理器中，在执行失败时，可以以明文的形式调试程序并将关于缺陷的反馈信息送到程序码供应商（程序销售商）。

本发明的第六个目的是提供一种微处理器，该微处理器可以以低成本高性能的形式实现上述第一至第五个目的。

为了实现第一个目的，本发明的第一个方面具有下列特征。通过提供读取功能的总线接口单元，制成单芯片或单封装的微处理器从微处理器外部的存储器（例如：主存储器）读取多个利用代码密钥加密

01.02.14

的程序。对于不同的程序，代码密钥不同。利用分别对应的解密密钥，解密单元对这些读出的程序进行解密，并且指令执行单元执行这些已解密程序。

在中断多个程序中一些程序的执行时，提供执行状态写入功能的上下文信息加密/解密单元利用对微处理器唯一的密钥对指明执行状态的信息进行加密直到中断程序的中断点和代码密钥出现，并将加密的信息作为上下文信息写入微处理器外部的存储器。

如果重新启动被中断的程序，提供重新启动功能的验证单元利用与微处理器的唯一密钥对应的唯一解密密钥解密上下文信息，并只有当包含在已解密上下文信息内的代码密钥（即：预定重新启动程序的代码密钥）与已中断程序的原始代码密钥一致时，才重新执行程序。

此外，为了实现第二和第三个目的，微处理器还具有：存储区（例如：寄存器），它在处理器的内部而且不能被读出到外部；加密属性写入单元（例如：指令 TLB），用于将程序的处理目标数据加密属性写入存储器。加密属性包括程序的代码密钥和加密目标地址范围。在上下文信息中至少含有一部分加密属性。

上下文信息加密/解密单元还将对微处理器唯一的、基于秘密信息的签名附加到上下文信息。这样，验证单元判别解密上下文信息内的签名是否与对微处理器唯一的、基于秘密信息的原始签名一致，如果一致，就重新启动已中断的程序。

同样，将加密程序中断点前的执行状态存储到外部存储器作为上下文信息，而将执行处理目标数据的保护属性存储到处理器内部的寄存器，因此，可以防止非法变更数据。

为了实现第四个目的，本发明的第二个方面具有下列特征。制成单芯片或单封装的微处理器在其内保持不能读出到外部的唯一秘密密钥。具有读取功能的总线接口单元事先从微处理器外部的存储器读取利用与秘密密钥对应的、微处理器的唯一公开密钥加密的代码密钥。具有第一解密功能的密钥解密单元利用微处理器的秘密密钥对读出的代码密钥进行解密。总线接口单元还从外部存储器读出多个利用分别

01.02.14

不同的代码密钥加密的程序。具有第二解密功能的代码解密单元对这些读出的程序进行解密。指令执行单元执行执行解密的程序。

如果中断多个程序中一些程序的执行，则随机数发生装置可以产生随机数作为临时密钥。上下文信息加密/解密单元将：第一数值，利用随机数，通过对指明中断程序的执行状态的信息进行加密获得；第二数值，利用中断程序的代码密钥，通过对此随机数进行加密获得；以及第三数值，利用微处理器的秘密密钥，通过对此随机数进行加密获得，写入外部存储器作为上下文信息。

如果重新启动执行程序，上下文信息加密/解密单元从外部存储器读出上下文信息，利用秘密密钥对上下文信息内的第三数值的随机数进行解密，并利用解密的随机数对上下文信息内的执行状态信息进行解密。同时，利用预定重新启动程序的代码密钥，对上下文信息内第二数值的随机数进行解密。将通过利用代码密钥解密第二数值获得的随机数和通过利用秘密密钥解密第三数值获得的随机数与临时密钥进行比较，并且仅在它们一致时，重新启动执行程序。

同样，利用在各存储时刻产生的随机数，将指明中断点时执行状态的上下文信息进行加密，并附加使用对微处理器唯一的秘密密钥的签名，因此，可以将上下文信息安全地存储到外部存储器。

为了实现第一至第三个以及第六个目的，本发明的第三个方面具有下列特征。制成单芯片或单封装的微处理器读出多个利用对不同程序不同的密钥加密的程序并执行它们。此微处理器具有不能读出到外部的内部存储器（例如：寄存器），此微处理器将将由各程序引用的数据（即处理目标数据）的加密属性和说明信息的加密属性存储到寄存器。上下文信息加密/解密单元将相关信息写入外部存储器，此相关信息与存储在寄存器内说明信息的并含有对微处理器唯一的签名的加密属性有关。根据程序提交的数据地址，保护表管理单元从外部存储器读取相关信息。利用秘密密钥，验证单元验证包含在所读出的相关信息内的签名。并且只有当此签名与对微处理器唯一的签名一致时，验证单元才根据说明信息的加密属性和读出的相关信息，允许程序引用

01.02.14

数据。

在这种配置中，待存储到内部寄存器的信息与签名附在一起并存储到外部存储器，并且只将必要部分读出到微处理器。在读取时验证签名，可以确保不受代换之害。即使当增加要处理的程序数并增加加密属性的数目时，也无需扩大微处理器内的存储区，因此可以降低成本。

根据本发明的一个方面，提供了一种微处理器，该微处理器具有与不能读出到外部的唯一秘密密钥对应的唯一秘密密钥和唯一公开密钥，它包括：读取单元，用于从外部存储区读出多个利用不同执行码密钥加密的程序；解密单元，被进行配置以利用各自解密密钥，对多个通过读取单元读出的程序进行解密；执行单元，被进行配置以执行多个通过解密单元解密的程序；上下文信息保存单元，被进行配置以将其执行被中断的程序的上下文信息保存到外部存储器或保存到在微处理器内部设置的上下文信息存储器，该上下文信息含有指明此程序的执行状态的信息和此程序的执行码密钥；以及重新启动单元，被进行配置以通过从外部存储器或上下文信息存储器读出上下文信息并通过从上下文信息中恢复此程序的执行状态，重新启动执行此程序。

通过以下结合附图的描述，本发明的其它特征和优势将会更加明显。

图 1 示出根据本发明第一实施例具有微处理器的系统的方框图。

图 2 示出在图 1 所示的微处理器内使用的全部存储空间的示意图。

图 3 示出根据本发明第二实施例的微处理器的基本配置的方框图。

图 4 示出图 3 所示的微处理器的详细配置的方框图。

图 5 示出在图 3 所示的微处理器中使用的页目录格式和页表格式的示意图。

图 6 示出在图 3 所示的微处理器中使用的页表格式和密钥输入格式。

图 7A 和图 7B 分别示出在图 3 所示的微处理器中使用的、交错前

01.04.19

和交错后的典型数据。

图 8 示出在图 3 所示的微处理器内执行的代码解密过程的信息流。

图 9 示出在图 3 所示的微处理器内使用的 CPU 寄存器。

图 10 示出在图 3 所示的微处理器内使用的上下文保持格式。

图 11 示出在图 3 所示的微处理器内执行的保护域切换过程的流程图。

图 12 示出在图 3 所示的微处理器内执行的数据加密和解密处理过程的信息流。

图 13 示出由图 3 的微处理器执行保护域内控制的处理的流程图。

图 14 示出概念性说明利用图 3 所示的微处理器的调用过程和从保护域转移到非保护域的过程的示意图。

图 15 示出在传统处理器内使用的上下文保存格式。

现在将参考图 1 和图 2 详细说明根据本发明的抗干预微处理器的第一实施例。

此第一实施例涉及一种微处理器，该微处理器用于防止程序指令秘密（执行码）和上下文信息（执行状态）被目标系统的用户使用，其中利用公开密钥（非对称密钥）加密系统以加密形式提供程序指令秘密和上下文信息。

图 1 示出目标系统，通过总线 2102，将目标系统的微处理器 2101 连接到主存储器 2103。

如图 1 所示，在此实施例中，微处理器 2101 具有寄存器文件 2111、指令执行单元 2112、指令缓冲器 2113、公开密钥解密功能块 2114、秘密密钥寄存器 2115、公共密钥解密功能块 2116、公共密钥寄存器 2117、BIU（总线接口单元）2118、缓冲寄存器 2119、公开密钥寄存器 2120、加密功能块 2121、解密功能块 2122 以及以前的公共密钥寄存器 2123，以下将对它们进行详细说明。

首先对将在下述描述中使用的术语进行说明，然后主要说明通用操作系统（OS）和应用程序的操作。程序是为特定目的编写的一组数据和一系列机器语言指令。OS 是用于管理系统资源的程序，而应用程

序是在 OS 资源管理的管理下运行的程序，该实施例预先支持多任务系统，因此，多个应用程序可以在 OS 的管理下以准并行的方式运行。以准并行方式运行的每个应用程序被称为进程。有时，将为相同目的执行进程的一组进程称为任务。

通常以文件的形式将应用程序的指令和数据存储到二级存储器。利用 OS 的装载程序，将它们设置到存储器，然后将它们作为进程执行。通常，利用输入/输出等引起的处理器异常处理（或中断）中断程序的执行。将执行异常处理的程序称为异常处理程序。通常利用 OS 安装异常处理程序。OS 可以处理硬件的异常请求、中断应用程序的运行过程并在任意时间重新启动或启动另一个应用程序。中断进程包括：无需在执行异常处理程序后切换进程，就重新启动原始进程的执行的暂时情况；以及要求切换进程的情况。前者的例子具有简单计时器而后者的例子具有由于页异常处理的虚拟存储器。

此实施例的目的是防止程序指令（执行码）和执行状态被目标系统用户使用，目标系统用户可以自由读目标系统主存储器并可以自由变更 OS 程序或应用程序。

实现此目的的基本特征是对处理器内信息存储的访问控制和根据下列信息的加密过程。

（1） 程序创建者选择的公共密钥 Kx，利用使用此密钥的秘密密钥加密系统对应用程序进行加密。

（2） 一对在处理器内设置的唯一公开密钥 Kp 和唯一秘密密钥 Ks。利用指令程序可以读出公开密钥。

（3） 密钥信息，其中的程序公共密钥 Kx 被利用处理器的公开密钥 Kp 加密。

[明文程序的执行]

此处理器可以执行具有共存明文指令和加密指令并被设置到主存储器的程序。这里将参考图 1 和图 2 所示的存储器分配，对在 CPU 内执行明文程序的运行过程进行说明。

图 2 示出整个存储器空间 2201，在存储器空间中，将程序设置到

主存储器上的区域 2202 至 2204，其中区域 2202 和 2204 为明文区域，而区域 2203 为加密区域。区域 2205 存储解密区域 2203 时使用的密钥信息。

当利用转移指令将控制从 OS 跳转到程序等的顶部 x 时，启动该程序的执行。指令执行单元 2112 执行跳转到 x 的指令，并将指令的地址输出到 BIU 2118。通过总线 2102 读取地址 x 的内容，并将地址 x 的内容从 BIU 2118 送到指令缓冲器 2113，然后送到指令执行单元 2112 执行该指令。其运行结果反应到寄存器文件 2111。当运行目标是对主存储器 2103 上的地址的读/写时，将其地址数值送到 BIU 2118，将此地址从 BIU 2118 输出到总线 2102，然后对存储器进行读/写操作。

指令缓冲器 2113 具有存储两条或多条指令的容量，并且从主存储器 2103 选择性地读出与指令缓冲器 2103 的容量对应的指令。

[加密指令的执行]

接着说明加密指令的执行情况。根据此实施例的处理器具有两种状态：明文指令的执行状态和加密指令的执行状态。因此提供了两种指令用于控制这两种状态。一种指令是加密执行启动指令，用于实现从明文指令的执行状态转换到加密指令的执行状态；另一个指令是明文返回指令，用于实现相反的转换。

[加密执行启动指令]

加密执行启动指令被表示为如下的助记符号“execenc”并具有一个操作数：

execenc keyaddr

其中“keyaddr”表示存储解密后续指令时使用的密钥信息的地址。

[密钥信息]

在此将说明密钥信息和程序加密过程。加密区域 2203 包括加密指令序列。将指令细分为以预取指令队列大小为单位的块，并利用诸如 DES（数据加密标准）算法的秘密密钥算法对指令进行加密。以下将此加密过程中使用的密钥表示为 Kx。由于使用了秘密密钥算法，所以解密时使用相同的密钥。

如果以明文的形式将此 Kx 设置到主存储器，则可以控制 OS 的用户可以容易地读取它并对加密程序进行分析。 为了防止这种情况发生，将通过利用处理器的公开密钥 Kp 加密 Kx 获得的 $E_{k\,p}$[Kx]设置到存储器区域 2205. "keyaddr" 表示区域 2205 的顶地址。

除非知道与公开密钥 Kp 对应的 Ks，否则不可能用密码方法（计算方法）对 $E_{k\,p}$[Kx]进行解密获得 Kx. 因此，只要目标系统的用户不知道 Ks，就一定不会将程序的秘密泄露给用户。在处理器内部，以不能由外部读取的方式存储此 Ks. 处理器可以在内部解密 Kx，而不会使用户得知此 Ks，而且处理器还可以利用 Kx 对加密程序进行解密并执行此程序。

以下将详细说明加密执行启动指令和后续的加密指令的执行. 通过执行区域 2207 内的转移指令，控制被转移到地址 "启动" 处的加密执行启动指令。在加密执行启动指令的操作数 "keyaddr" 指明的地址，将规定区域 2205 的内容作为数据读出到处理器的指令执行单元 2112. 指令执行单元 2112 将此数据 $E_{k\,p}$[Kx]送到公开密钥解密功能块 2114. 通过利用在秘密密钥寄存器 2115 存储器储的、对处理器唯一的秘密密钥 Ks 解密 $E_{k\,p}$[Kx]获得 Kx. 并将它存储到公共密钥寄存器 2117. 然后，处理器进入加密指令执行状态。

在此，假设这样制造处理器封装，以致不能利用处理器芯片的程序和调试程序将秘密密钥寄存器 2115 和公共密钥寄存器 2117 存储器储的内容读出到外部。

通过执行加密执行启动指令，将解密后续指令使用的密钥存储到公共密钥寄存器 2117，然后处理器进入加密指令执行状态。当处理器处于加密指令执行状态时，将从主存储器 2103 读取的指令由 BIU 2118 送到公共密钥解密功能块 2116，并利用存储在公共密钥寄存器 2117 内的密钥信息对它进行解密，然后将它存储到指令缓冲器 2113.

在此实施例中，紧跟在加密执行启动指令之后存储到区域 2204、利用密钥 Kx 加密的程序被解密并被存储到指令缓冲器 2113，然后执行它。以指令缓冲器 2113 的大小为单位进行读取。图 2 示出指令缓冲

01-02-14

器 2113 的大小为 64 位的典型情况，并且选择性地将 16 位大小的四条指令分别读出到指令缓冲器 2113.

[明文返回指令]

通过执行明文返回指令，处于加密指令执行状态的处理器返回明文指令执行状态。

明文返回指令被表示为如下助记符号:

exitenc

它没有操作数. 通过执行此指令，经过不通过公共密钥解密功能块 2116 的通路从主存储器 2103 读取该指令，然后处理器返回明文指令执行.

请注意，当在加密指令执行期间再执行加密执行启动指令时，改变指令解密密钥，以致可以利用不同密钥解密后续指令，然后执行后续指令.

[上下文保存和对其的攻击]

接着将说明为了在多任务环境下保护应用程序的秘密而安全保存执行状态的情况.

此处理器的寄存器文件 2111 具有 32 个通用寄存器（R0 至 R31）. R31 用作程序计数器. 将通用寄存器的内容存储到寄存器文件 2111. 当在上述加密程序的执行期间发生异常时，将寄存器文件 2111 的内容转移到缓冲寄存器 2119, 并用预定数值或随机数初始化寄存器文件 2111 的内容. 然后，将用于解密加密程序的公共密钥数值存储到前一个公共密钥寄存器 2123. 只有完成这两种初始化之后，才可以将控制转移到异常处理程序并执行异常处理程序的指令. 假设异常处理程序的指令未加密.

利用此寄存器文件的初始化功能，在此实施例的处理器中，即使是在由于加密程序的执行期间发生异常而将控制转移到异常处理程序的情况下，仍可以防止读取加密程序利用异常处理程序处理的寄存器数值. 同时，将寄存器文件 2111 的内容保存到缓冲寄存器 2119. 以下将说明为了重新启动加密程序，用于恢复缓冲寄存器内容并用于将它

们存储到存储器的功能块。

可以直接由异常处理程序的非加密程序读出存储在缓冲寄存器2119 的寄存器内容。异常处理程序的非加密程序只允许对缓冲寄存器2119进行如下两步操作。

（1） 恢复缓冲寄存器内容并重新启动执行原始加密程序。

（2） 加密缓冲寄存器的内容并将它们存储到存储器，然后执行OS 程序或其它加密程序。

在（1）操作情况下，当处理诸如计数器递增的异常处理程序完毕后，异常处理程序发出“cont”（继续）指令。当执行“cont”指令时，在寄存器文件2111和公共密钥寄存器2117内分别恢复缓冲寄存器2119的内容和前一个公共密钥寄存器2123的内容。由于在寄存器文件2111内含有程序计数器，因此通过使控制退回到中断加密程序执行的点，可以重新启动加密程序的执行。为了在重新启动之后对加密程序解密，可以使用从前一个公共密钥寄存器 2123 恢复的数值。与缓冲寄存器2119的内容相同，显然，该程序不能重写前一个公共密钥寄存器2123。

（2）操作情况与在执行异常处理程序定时发生的进程切换的情况对应。在这种情况下，处理器的异常处理程序或任务调度程序发出“savereg”（保存寄存器）指令用于将缓冲寄存器 2119 的内容保存到存储器。此“savereg”指令被表示为如下助记符号：

savereg dest

并且该指令具有一个操作数“dest”，操作数“dest”代表保存缓冲寄存器内容的地址。

发出“savereg”指令时，通过使用存储在公开密钥寄存器 2120内的处理器公开密钥 Kp，利用加密功能块2121对缓冲寄存器2119和前一个公共密钥寄存器 2123 的内容进行加密，并通过 BIU2118 将它们保存到主存储器 2103 内由“dest”规定的地址。主存储器 2103 在处理器的外部，因此有被用户访问的可能性，但是利用处理器的公开

密钥可以将这些内容加密，这样不知道处理器的秘密密钥的用户不可能得知缓冲寄存器的内容。

保存缓冲寄存器的内容后，利用上述方法，OS 激活另一个加密程序。如果未保存缓冲寄存器的内容就激活另一个加密程序，则当中断另一个加密程序的执行时，会将缓冲寄存器的内容重写到另一个加密程序的缓冲寄存器，并且由于原始加密程序已丢失，所以不可能将原始加密程序作为缓冲寄存器内容重新启动。

在此，假定缓冲寄存器的个数为 1，但是为了处理多个异常，也可以具有多个缓冲寄存器。

[恢复过程]

接着，将说明已保存执行状态的恢复过程。

在重新启动被中断应用程序时，OS 的调度程序发出“rcvrreg”（恢复寄存器）指令。此“rcvrreg”指令被表示为如下助记符号：

rcvrrdg addr

并且该指令具有一个操作数“addr”，该操作数“addr”代表保存执行状态的地址。

发出“rcvrreg”指令指令时，利用处理器 BIU 2118，从“addr”规定的存储器地址取出加密执行状态信息，利用解密功能块 2122，通过使用处理器的秘密密钥 Ks 对它进行解密，然后在寄存器文件 2111 内恢复寄存器信息，而在公共密钥寄存器 2117 内恢复程序解密密钥。恢复完成时，从程序计数器指明的点重新启动已中断程序的执行。此时，将从执行状态信息中恢复的密钥 Kx 用于加密程序的解密过程。

以上说明了由于异常中断的加密程序的执行状态的保存过程和恢复过程的细节。如上所述，加密程序可以避免受到可以控制目标系统 OS 用户的攻击。

接着，将说明防止两种攻击执行状态的方式的上述方法的安全性。

[攻击执行状态]

01.02.14

有两种方式可以攻击应用程序执行中产生的执行状态。一种方式是利用攻击者窥视保存的执行状态，而另一种方法是利用攻击将执行状态重写为要求数值。

在此，将定义如下两个用于解释非法访问执行状态的术语。首先，将产生执行状态的程序称为此执行状态的原始程序。通过在寄存器内恢复执行状态，可以重新启动原始程序。另外，将产生执行状态的程序以外的程序（即利用不同于原始程序的密钥或明文程序的密钥的密钥加密的程序）称为其它程序。

将对某些原始程序产生的执行状态的非法访问或攻击定义为，不知道原始程序密钥的第三方利用某些独立于处理器操作的方法对存储器上的执行状态进行直接分析的行为，或第三方利用在相同处理器上运行的其它程序分析执行状态或将执行状态重写为要求的数值的行为。

在此实施例的微处理器中，利用如下三种机制可以保护执行状态，这样就可以防止利用访问处理器外的存储器或利用其它程序进行非法访问。

首先，在此实施例中，当加密程序的执行被中断时，将寄存器信息保存到缓冲寄存器 2119. 然后，利用“rcvrreg”指令或“savereg”指令方法之外的任何方法均不能访问缓冲寄存器 2119 和前一个公共密钥寄存器 2123，所以，其它程序不能自由读取它们的内容。

在传统处理器中，利用异常处理程序可以自由读取发生异常时的寄存器内容。在此实施例的微处理器中，寄存器内容被保存到缓冲寄存器 2119，因此可以禁止其它程序读取寄存器内容，为了防止系统用户窥视存储在存储器上的执行状态，提供用于通过利用处理器的公开密钥对它们进行加密来保存缓冲寄存器内容的指令。

第二种攻击方法是，通过在与原始程序相同的存储器地址设置为攻击者所知的某些其它程序指令来读取包含在执行状态内的寄存器数值，以致使此其它程序读取加密执行状态。

在此实施例的微处理器中，加密执行状态含有程序密钥，并且在

重新启动时将此密钥用于解密加密程序。由于有此机制，即使在原始程序之外的其它程序试图读取执行状态时，由于密钥不匹，所以不能将程序直接解密并且不能按照攻击者的意图执行程序。这样，在此实施例的微处理器中就不可能使用第二种攻击方法。

通过利用处理器的公开密钥简单解密执行状态本身不能实现此效果，但是通过对原始程序的密钥和执行状态整体进行加密可以实现此效果。

请注意，为了使此效果最好，在使用公开密钥进行加密时，优先将寄存器（R0 至 R31）的数值和公共密钥 Kx 存储到相同的密码块。

[数据保护]

在此实施例的微处理器中，未考虑对数据进行加密。但是，对于本技术领域的技术人员来说显而易见，与在微处理器中进行数据加密用以支持将在第二实施例中说明的虚拟存储器相同，可以将数据加密功能添加到此实施例的微处理器。

现在参考图 3 至图 14，详细说明根据本发明抗干预微处理器的第二实施例。

在此实施例中，对于使用基于 Intel 公司推出的、广泛使用的 Pentium Pro 微处理器结构的典型情况，说明根据本发明的微处理器，但是本发明并不局限于此特定结构。在以下的说明中，将说明 Pentium Pro 微处理器结构的具体特征并将说明使用其它结构的情况。

请注意，Pentium Pro 结构对地址空间划分了三种地址，它们包括：物理地址、线性地址以及逻辑地址，但是在此实施例中将 Pentium 术语中的线性地址称为逻辑地址。

在以下的说明中，除非另有说明，保护包括对应用程序秘密的保护（即利用加密进行保护）。因此，应清楚地将此实施例中的保护与通常使用的保护概念区别开，这是预防由于某些程序的运行对其它应用程序运行的干扰。然而，在本发明中，在普通意义上，当然由 OS 提供运行保护机制（由于它与本发明无关，所以省略了对这方面的说明），该保护机制与根据本发明应用程序的秘密保护并行。

01.02.14

此外，在以下说明中，将处理器可以执行的机器语言指令称为指令，并且选择性地将多条指令称为执行码或指令流。将加密指令流过程使用的密钥称为执行码密钥。

此外，在以下说明中，将秘密保护机制称为在 OS 管理下的应用程序保护秘密，但是可以将此机制用作防止 OS 本身被变更或分析的机制。

图 3 示出根据此实施例的微处理器的基本配置。图 4 示出图 3 所示的微处理器的详细配置。

微处理器 101 具有：处理器核心 111、指令 TLB（查表缓冲器）121、异常处理单元 131、数据 TLB（查表缓冲器）141、二级高速缓存 152。处理器核心 111 包括总线接口单元 112、代码和数据加密/解密处理单元 113、一级高速缓存 114 以及指令执行单元 115。指令执行单元 115 进一步包括指令提取/解码单元 214、指令表 215、指令执行转换单元 216 以及指令执行完成单元 217。

异常处理单元 131 进一步包括寄存器文件 253、上下文信息加密/解密单元 254、异常处理单元 255、秘密保护破坏检测单元 256 以及执行码密钥与签名验证单元 257。

指令 TLB 121 进一步包括页表缓冲器 230、执行码解密密钥表缓冲器 231 以及密钥单元 232。数据 TLB 141 进一步包括保护表管理单元 233。

微处理器 101 具有用于存储对此微处理器唯一的公开密钥 Kp 和秘密密钥 Ks 的密钥存储区 241。现在研究从某些程序销售商购买要求的执行程序 A 并执行它的情况。程序销售商在提供执行程序 A 之前利用公共执行码密钥 Kcode（$E_{Kcode}[A]$）将程序 A 加密，然后将用于以利用微处理器 101（$E_{Kp}[Kcode]$）公开密钥 Kp 的加密方式进行加密的公共密钥 Kcode 送到微处理器 101。微处理器 101 是一种多任务处理器，它不仅可以处理执行程序 A 而且可以以准并行方式处理多个不同的加密程序（即通过允许中断实现）。此外，微处理器 101 可以预先执行加密程序和明文程序。

01.02.14

通过总线接口单元（读取功能块）112，微处理器 101 从微处理器 101 外部的主存储器 281 读出多个利用不同的执行码密钥加密的程序。利用各自对应的解密密钥，执行码解密单元 212 对此多个读出程序进行解密，然后，指令执行单元 115 执行此多个解密程序。

在中断一些程序的执行的情况下，利用微处理器的公开密钥，异常处理单元 131 的上下文信息加密/解密单元 254 将指明被中断程序中断点处执行状态和此程序的代码密钥的信息加密，然后将此加密信息作为上下文信息写入主存储器 281。

在重新启动中断程序的情况下，利用微处理器 101 的秘密密钥，执行码密钥与签名验证单元 257 将加密上下文信息解密，验证解密上下文信息内的执行码密钥（即：预定重新启动程序的执行码密钥）是否与中断程序的原始执行码密钥一致，只有在它们一致时才重新启动该程序的执行。

在说明微处理器 101 的详细配置和功能之前，这里先概括说明利用微处理器 101 对明文指令执行和执行加密程序的处理过程。

当微处理器 101 执行明文指令时，指令提取/解码单元 214 试图从 L1 指令高速缓存 213 读取由程序计数器（未示出）指明的地址上的内容。如果规定地址上的内容被高速缓存，则从 L1 指令高速缓存 213 读出程序并将此程序送到指令表 215，然后执行它。指令表 215 可以并行执行多条指令，请求将完成执行的必要数据读到指令执行转换单元 216，然后接收此数据。当并行执行指令并且确定它们的执行结果时，将执行结果送到指令执行完成单元 217。当运行目标是微处理器 101 内的寄存器时，指令执行完成单元 217 将执行结果写入寄存器文件 253；当执行目标是存储器时，指令执行完成单元 217 将执行结果写入 L1 数据高速缓存 218。

L1 数据高速缓存 218 的内容在总线接口单元 112 的控制下被 L2 高速缓存 152 再一次高速缓存，并被写入主存储器 281。这里使用了虚拟存储器机制，图 5 所示的页表说明逻辑存储地址与物理存储地址有相似之处。

01.02.14

页表是一种设置到物理存储器的数据结构。实际上，数据 TLB141 实现从逻辑地址到物理地址的转换，同时管理数据高速缓存。根据微处理器 101 内的寄存器指明的页表顶地址，数据 TLB 读取页表上的必要部分，并执行将逻辑地址转换到物理地址的操作。此时，根据将访问的逻辑地址，只将页表的必要部分读出到页表缓冲器 234，而不是将存储器上的整个页表读出到数据 TLB 141。

基本高速缓存操作稳定，与程序指令是否被加密无关。换句话说，将部分页表读出到指令 TLB 121，并根据这里给出的解释进行地址转换。总线接口单元 112 从主存储器 281 或 L2 高速缓存 152 读取指令，并将这些指令存储到 L1 指令高速缓存 213。将指令读出到 L1 指令高速缓存 213 是以由多个字组成的行单位实现，这样进行访问比以字为单位进行访问快。

利用物理存储器上的相同页表对已执行指令的处理目标数据进行地址转换，如上所述，在数据 TLB 141 进行这种转换。

此点前的操作基本上与一般高速缓冲存储器相同。

接着将说明执行加密程序情况的运行过程。在此实施例中，假设秘密被保护的执行码全部被加密，并且将加密的执行码称为保护码。此外，将相同密钥的保护范围称为保护域。换句话说，由相同密钥保护的代码集属于相同保护域，而由不同密钥保护的代码属于不同保护域。

首先，利用秘密密钥方法块密码算法加密的程序的执行码被存储到主存储器 281。以下将说明装入程序销售商发送的加密程序的方法。

执行码的密码块大小可以是任意值，只要 2 的密码块大小次方与作为读/写高速缓冲存储器单位的行的大小一致就可以。然而，如果块

大小太小以致块长度与指令长度相同，就产生了容易通过记录加密数据与可预测指令部分（例如：子例程的顶端部分）的相似之处分析指令的可能性。为此，在此实施例中，块被交错，以致在块内数据之间存在互相依赖性并且加密块含有关于多个指令字或操作数的信息。同样，难于使指令与加密块相符。

图 7A 和图 7B 示出可以用于此实施例的交错的实例。在此例中，假设高速缓存的行大小为 32 字节，块大小为 64 位（即 8 个字节）。如图 7A 所示，在进行交错前，一个字由 4 个字节组成，因此，字 A 由 A0 至 A3 的四个字节组成。一行由 A 至 H 的 8 个字组成。当这是以对应于 64 位块大小的 8 个字节为单位进行交错时，如图 7B 所示，将 A0、B0、....、H0 安排到对应于字 0 和字 1 的第一块，将 A1、B1、....、H1 安排到下一个块，等等。

通过使交错的区域长度更长，可以使攻击更难，但是由于一个高速缓存行的解密/加密依赖于另一行的读/写，所以具有比行大小更长长度的区域的交错使得处理更复杂了并降低了处理速度。因此优先在高速缓存行大小范围内设置交错范围。

使用交错块数据的方法，会在高速缓存行内的多个块的数据之间存在互相依赖性，但是还可以使用能在数据块之间产生依赖性的其它方法（例如：块密码的 CBC（密码块链接）方式）。

根据页表确定加密执行码的解密密钥 Kcode（即使在解密情况下，以下也将它称为加密密钥，因为在秘密密钥算法中，加密密钥与解密密钥相同）。图 5 和图 6 示出从逻辑地址转换到物理地址的表结构。

程序计数器的逻辑地址 301 列出一些数值，构成其高位的目录 302 和表 303 确定页入口 307-j。页入口 307－j 含有密钥入口 ID 307－j－k，而在密钥表 309 内根据此 ID 确定用于对此页进行解密的密钥入口 309－m。用微处理器内的密钥表控制寄存器 308 存储密钥表 309 的物理地址。

在此实施例中，将密钥入口的 ID 设置到页入口而不是直接设置密钥信息，这样为了节省指令 TLB121 上的有限存储区空间，在多个页

之间共享大量的密钥信息。

如下所述更详细地将页表和密钥表信息存储到指令 TLB 121。只将访问存储器所必须的部分从页表 306、307 和 311 读出到页表缓冲器 230，从页表 309 读出到执行码解密密钥表缓冲器 231。

在存储到主存储器的状态中，作为密钥表 309 的一个元素的密钥对象 309-m 基准计数器指明涉及此密钥对象的页表数。在将密钥对象读出到执行码解密密钥表缓冲器 231 的状态中，此基准计数器指明涉及此密钥对象并被读出到页表缓冲器 230 的页表数。在从执行码解密密钥表缓冲器 231 删除不必要的密钥对象时，此基准计数器用于判断。

此实施例的特征之一是密钥表入口具有固定长度，但是为了能够对付更高的密码分析能力，可以改变各表中使用的密钥长度，并在密钥表的密钥大小区说明在各表使用的密钥长度。这意谓着对微处理器 101 唯一的秘密密钥 Ks 固定，但是利用密钥入口的说明，可以改变用于程序加密与解密的 Kcode 的长度。为了说明可变长度密钥的位置，密钥入口 309-m 具有指向密钥入口的字段 309-m-4，字段 309-m-4 指明密钥对象的地址。

在密钥对象区域 310 中，以使用微处理器 101 公开密钥 Kp 的公开密钥算法加密的 E_{kp}[Kcode]的形式存储执行码密钥 Kcode。为了以公开密钥算法的形式对数据可靠加密，必须有大冗余，因此，加密数据的长度比原始数据的长度长。这里将 Ks 和 Kp 的长度设置为 1024 位，将 Kcode 的长度设置为 64 位，通过填充可以将它扩展到 256 位，然后将 E[Kcode]加密成 1024 位并存储到密钥对象区 310。当 Kcode 具有这样的长度以致不能以 1024 位存储它时，就将它细分为多个分别是 1024 位大小的块，然后存储。

图 8 是以代码解密方式概括说明的信息流。程序计数器 501 指明在逻辑地址空间 502 上的加密代码区 502 的地址“Addr”。根据被读出到指令 TLB 121 的页表 307，将逻辑地址“Addr’”转换为物理地址“Addr”。同时，将加密代码解密密钥 E[Kcode]从密钥表 309 中取出，利用在 CPU 内提供的秘密密钥 Ks，解密功能块 506 对它进行解

密，然后将它存储到当前代码解密密钥存储单元 507. 程序销售商利用微处理器 101 的公开密钥 Kp 将用于代码加密的公共密钥 Kcode 加密，并与利用 Kcode 加密的程序一起提供公共密钥 Kcode，因此不知道微处理器 101 的秘密密钥 Ks 的用户就不可能知道 Kcode。

利用 Kcode 将程序执行码加密并发运之后，程序销售商保持并控制 Kcode 的安全性，以致不会将其秘密泄露给第三方.

将整个密钥表 511 和整个页表 512 设置到物理存储器 510，并分别用密钥表寄存器 508 和 CR3 寄存器 509 来存储它们的地址. 根据整个密钥表 511 和整个页表 512，通过总线接口单元 112，只将必须部分高速缓存到指令 TLB 121.

通过总线接口单元 112 将对应于利用指令 TLB 121 转换的物理地址“Addr’”的内容 503 读出时，将此页加密，这样，就可以在代码解密功能块 212 将内容 503 解密。以高速缓存行大小为单位进行读，并在块单元内进行解密后，进行上述交错过程的反过程。将解密结果存储到 L1 指令高速缓存 213，然后作为指令执行.

这里将说明载入加密程序和重定位加密程序的方法。为了将程序载入存储器，一种方法是，程序载入器改变程序执行码含有的地址值以处理由于载入程序引起的地址变化，但是这种方法不能应用于加密程序。然而，利用实现重定位的方法，可以将加密程序重定位，而无需利用被跳转地址表调用的表或 IAT（输入地址表）直接重写执行码.

载入过程和对一般程序重定位的进一步详情可以参考，例如载入方法和加密程序的重定位可以参考本申请人申请的第 2000－35898 号日本专利申请.

利用上述解密程序的加密执行码并将它们读出到处理器内的高速缓冲存储器然后执行它们的方法，可以保护设置到处理器外部存储器上的执行码.

然而，在处理器内可以存在被解密为明文的执行码. 即使不可能直接从处理器外部将它们读出，但是存在明文程序被在相同处理器内

运行的其它程序读出并分析的可能性。

在此实施例中，在将数据读入 L1 数据高速缓存 218 时，未进行利用指令 TLB 241 内的秘密密钥 241 和密钥解密单元 232 的密钥解密处理过程。当对在页表中将加密标志 307－j－E 设置为“1”的加密页读取数据时，或者读出非解密原始数据或者读出规定数值“0”的数据，否则就发生异常不能读出正常解密的数据。请注意，当页表内的加密标志 307－j－E 被重写时，相应指令高速缓存的解密内容将会失效。

利用此机制，其它程序（包括专用程序）不可能读取加密程序的执行码作为数据，并且不可能利用处理器的功能对它们进行解密。

此外，其它程序显然不能读取指令高速缓存内的数据，因此可以保证执行码的安全性。以下将说明数据的安全性。

由于可以用同样的方法执行加密执行码，因此，在此实施例的微处理器中，通过适当选择加密算法和参数，用密码方法可以使不知道执行码加密密钥 Kcode 真实值的一方不能通过反汇编执行码来分析程序的操作过程。

因此，用户不可能知道执行码加密密钥 Kcode 的真实值，并且用密码的方法使用户不能根据用户的意图进行变更（例如：通过变更部分加密程序，非法复制应用程序处理的内容）。

接着，将说明此实施例微处理器的另一个特征，该特征涉及加密、签名及其在多任务环境下中断程序执行时对上下文的验证。

经常由异常来中断多任务环境下程序的执行。通常，当执行被中断时，将处理器的状态存储到存储器，然后在后来重新启动此程序的执行时恢复原始状态。同样，可以以准并行方式执行多个程序并认可中断处理过程。中断时的状态信息被称为上下文信息，上下文信息含有应用程序使用的寄存器信息，此外还含有在一些情况下不是应用程序明确使用的寄存器的信息。

在传统处理器中，当一些程序在执行期间发生中断时，将控制转移到 OS 的执行码，而保持应用程序的寄存器状态，因此，OS 可以校验该程序的寄存器状态以推测执行了什么指令，或在中断期间变更以

明文形式保持的上下文信息，因此，在重新启动该程序的执行后，可以改变程序的运行。

根据此事实，在此实施例中，当在保护代码的执行期间发生中断时，在此之前立即将执行的上下文信息加密或存储，而所有的应用程序寄存器或者被加密或者被初始化，并将处理器实现的签名附到上下文信息。在从中断恢复时验证签名以校验签名是否正确。当检测到不正确的签名时，就停止恢复，这样就可以防止用户变更上下文信息。此时，加密目标寄存器即图 9 所示的用户寄存器 701 至 720.

在 Pentium Pro 结构中，存在支持将处理过程的上下文信息保存到存储器并将其恢复的硬件机制。保存状态的区域被称为 TSS（任务状态段）。以下将说明将本发明应用于此机制的典型情况，但是本发明并不局限于 Pentium Pro 结构，并且通常同样可以应用于任何处理器结构。

在下列情况下，保存上下文和异常发生一起进行。当发生异常时，从称为 IDT（中断描述表）的表中读出对应于中断原因的入口用于描述异常处理过程，并执行在此描述的异常过程。当入口指明 TSS 时，将在所指明的 TSS 内保存的上下文恢复到处理器。反之，将直到那时执行的过程的上下文保存到任务寄存器 725 此时规定的 TSS 区。

利用此自动上下文保存机制，可以将包括程序计数器和堆栈指针在内的应用程序的全部状态保存，并通过验证签名来检测恢复时的变更。然而，当使用此自动上下文保存时，除了由上下文切换产生大开销之外，还会产生没有使用 TSS 就不能实现中断处理的问题。

为了减少由于中断处理产生的开销，或为了保持与现有程序的兼容性，优先不使用自动上下文保存机制。但是在这种情况下，将程序计数器保存到堆栈而且不能作为验证目标，因此，它可以是被恶性 OS 变更的目标。应优先根据目的正确利用这两种情况。为此，由于更重视安全性，此实施例的微处理器对被保护的（被加密的）执行码采取自动上下文保存。不必所有的寄存器均是自动保存寄存器。

在此实施例中，上下文保存过程和恢复处理过程具有如下三个主

要特征。

（1） 只有产生上下文的微处理器和知道产生上下文的程序的加密密钥 Kcode 的人才可以将已保存的上下文内容解密。

（2） 在一些执行码密钥 X 保护的程序被中断以及其上下文被保存的情况下，不能将其重新启动处理过程应用于非保护程序或利用另一个执行码密钥 Y 加密的程序的重新启动过程。换句话说，在重新启动时，利用另一个程序替换将由中断恢复的程序。

（3） 禁止恢复被变更的上下文。换句话说，如果变更被保存的上下文，则该上下文不能被恢复。

利用上述特征（1），可以保持上下文信息的安全性，同时允许程序销售商分析上下文信息。为了通过根据用户使用程序的条件来分析产生故障的原因从而维护程序的质量，重要的是程序销售商具有分析上下文的权利。上述特征（2）能够有效防止攻击者为了分析程序 B 内的数据秘密或代码秘密或者变更程序 B 的运行，将通过执行程序 A 产生的上下文应用于另一个加密程序 B 并从上下文中保持的已知状态重新启动程序 B。此功能还是如下所述的数据保护的先决条件，在数据保护中，多个应用程序中的各应用程序保持自有加密数据专用并独立于其它数据。

利用上述特征（3），可以严格避免利用重新启动程序的机会变更上下文信息。

提供这种功能的原因是为了根据处理器的秘密信息简单加密上下文信息可以防止按照攻击者的意图变更上下文信息，但是不可能避免由于随机错误重新启动程序导致随机变更上下文的可能性。

以下将详细说明具有上述三个特征的上下文保存和验证方法。

<上下文保存处理过程>

图 10 概念性示出根据此实施例的上下文保存格式。假设在保护程序的执行期间由于相关硬件原因或软件原因发生中断。如果对应于中断的 IDT 入口指明 TSS，则此点的程序执行状态被加密，并被作为上下文信息保存到由当前任务寄存器 725 指明（而不是由 TSS 本身指明）

01.02.14

的 TSS. 然后，将保存在 IDT 入口指明的 TSS 内的执行状态恢复到处理器。如果 IDT 入口未指明 TSS，则只进行当前寄存器的加密和初始化，而且不会发生保存到 TSS. 当然，在这种情况下不可能重新启动该程序。然而，请注意，包括一部分标志寄存器和任务寄存器在内的系统寄存器被排除在寄存器的加密或初始化目标之外，以便继续 OS 运行.

实际上，图 10 所示的上下文内容是在块单元内被交错、加密，然后被存储到存储器. 这里首先说明待存储的信息入口. 在顶端，提供对应于各自特权描述的堆栈指针和用户寄存器 802 至 825，然后接着设置指明 TSS 大小和加密的存在/不存在的一个字 826. 该字指明保存寄处理器的 TSS 是否被加密. 即使在 TSS 被加密的情况下，此区域仍保持明文形式，无需进行加密.

此后，设置为了进行数据保护而附加的数据加密控制寄存器（CY0 至 CY3）区域 827 至 830，并设置用于将大小调节到块长度的填充 831. 最后，设置：数值 E_{Kcode}[Kr] 832，利用使用执行码加密密钥 Kcode 的秘密密钥算法将其用于加密上下文的密钥 Kr 加密；数值 E_{Kp}[Kr] 833，利用处理器的公开密钥 Kp 将其用于加密上下文的密钥 Kr 加密；以及签名 S_{Ks}[message] 834，对它们使用处理器的秘密密钥 Ks. 此外，为了使 OS 能够进行任务调度，以明文形式将用于链接到先前任务的、在任务之间保持调用关系的区域 801 保存.

图 4 所示的异常处理单元 131 内上下文信息加密/解密单元 254 实现这些执行码加密过程和签名产生过程，它基于独立于执行码处理目标数据的加密过程的功能. 在 TSS 内保存上下文信息时，即使利用其它数据加密功能在 TSS 导致规定了一些加密，但是可以忽略此规定并且以上下文加密的状态保存上下文信息. 这是因为对各保护（加密）程序规定数据加密功能的加密属性，因此一些程序的重新启动不能依赖于此功能.

在加密上下文过程中，将以明文形式记录的 TSS 大小区域 826 内的字被复位为数值“0”. 然后，进行与参考图 7A 和 7B 解释的交错相

01.02.14

同的交错，并将上下文加密。此时，这样设置填充 831 的大小以致可以根据加密块的大小进行适当交错。

不利用处理器的公开密钥 Kp 或执行码加密密钥 Kcode 直接对寄存器进行加密的原因是为了允许程序销售商和处理器分析加密的上下文，而禁止用户对上下文进行解密。

由于程序销售商知道执行码加密密钥 Kcode，所以程序销售商通过利用 Kcode 解密 $E_{Kcode}[Kr]$可以获得上下文加密密钥 Kr。此外，通过利用专用秘密密钥 Ks 解密 $E_{Kp}[Kr]$，微处理器 101 可以获得上下文加密密钥 Kr。换句话说，程序销售商无需知道用户微处理器的秘密密钥通过解密上下文信息就可以分析故障，并且微处理器 101 本身通过利用自有秘密密钥 Ks 解密上下文信息可以重新启动执行。不具有这两个密钥的用户不能将保存的上下文信息解密。此外，不知道微处理器 101 的秘密密钥 Ks 的用户不可能对 $E_{Kcode}[Kr]$和 $E_{Kp}[Kr]$伪造上下文信息和签名 $S_{Ks}[message]$.

为了使程序销售商和微处理器互相独立地对上下文信息进行解密，还可以考虑一种直接利用 Kcode 加密上下文信息的方法。然而，在已知寄存器状态的情况下，存在已知明文对执行码加密密钥 Kcode 攻击的可能性。换句话说，当用于加密数据的密钥数值固定时，就会产生下列问题。现在研究执行读取用户输入数据并通过对它进行加密临时将它写入工作存储器的程序的情况。通过监视存储器，可以确定将被加密并将被写入工作存储器的数据，因此，用户通过改变输入值可以重复输入多次并可以获得相应的加密数据。这意谓着，密码分析理论的选择明文攻击法是可能的。

已知明文攻击对秘密密钥算法并不是致命的，但是，仍然优先避免这种情况。为此，在每次保存上下文时，异常处理单元 131 内的随机数发生装置 252 产生随机数 Kr，并将它送到上下文信息加密/解密单元 254。通过使用随机数 Kr 的秘密密钥算法，上下文信息加密/解密单元 254 将上下文加密。然后，附加数值 $E_{Kcode}[Kr]$ 832，通过同样的使用执行码密钥 Kcode 的秘密密钥算法可以将数值 $E_{Kcode}[Kr]$ 832

内的随机数 Kr 加密。通过使用微处理器的公开密钥 Kp 的公开密钥算法对随机数 Kr 进行加密可以获得数值 E_{Kp}[Kr] 833。

这里，随机数由随机数发生装置 252 产生。在程序被加密的情况下，通常程序码不发生变化，因此只要运行不被分析，就不可能非法获得相应的明文代码。既然是这样，为了进行密码分析就需要进行“仅知密文攻击法”攻击，因此，非常难以发现加密密钥。然而，在用户输入的数据将被通过加密存储到存储器的情况下，用户可以自由选择输入数据。为此，用户可以对密钥进行比“仅知密文攻击法”更有效的“选择明文攻击法”攻击。

为了防止选择明文攻击，可以采用一种措施通过将称为“盐”的随机数附加到待保护的明文将搜索空间加大。然而，以在应用编程层将“盐”随机数插入各数据中的方式保存到存储器很慢，因此，这样会降低编程的效率和效能。

为此，随机数发生装置 252 产生保存上下文时用于加密上下文的随机数（密钥）。由于可以任意选择密钥。因此可以在进程与进程之间或进程与设备之间实现更快速、安全、有效通信。这是因为，在存储器访问时用硬件加密数据的速度比用软件加密数据的速度慢得多。

相反，如果数据区域内的密钥数值被限制到规定的数值（例如：与执行码密钥相同的数值），这样就不可能对被利用其它密钥加密的或与设备共享加密数据的其它程序使用处理器的数据加密功能，因此，不可能利用处理器提供的快速硬件加密功能。

请注意，重新启动时进行的加密随机数 E_{kcode}[Kr] 832 的解密过程和签名 834 的产生过程可以根据任何算法和秘密信息进行，只要满足条件：仅由微处理器可以实现加密随机数 E_{kcode}[Kr] 832 的解密过程和签名 834 的产生过程即可。在上述实例中，对微处理器 101 唯一的秘密密钥 Ks（它还用于执行码加密密钥 Kcode 的解密过程）用于加密随机数 E_{kcode}[Kr] 832 的解密过程和签名 834 的产生过程，但是对这两种用途可以分别使用不同的数值。

此外，保存的上下文含有指明存在/不存在加密的标志，因此，根

据需要，加密上下文信息和非加密上下文信息可以共存．以明文形式存储 TSS 的大小和指明存在/不存在加密的标志，因此可以容易地保持与过去程序的兼容性。

<重新启动中断程序的处理过程>

通过恢复上下文重新启动进程时，OS 发布关于指明保存的 TSS 的 TSS 描述符的转移指令或调用指令。

参考图 4，首先，利用处理器的秘密密钥 Ks，异常处理单元 131 内的执行码密钥与签名验证单元 257 验证签名 S_{Ks}[message] 834，然后将验证结果送到异常处理单元 255. 验证结果失败时，异常处理单元 255 停止重新启动执行程序并产生异常．利用此验证，可以确认上下文信息确实是由具有秘密密钥的适当微处理器 101 产生的，并且未被变更．

当验证签名成功时，通过利用秘密密钥 Ks 解密上下文密钥 E_{Kp}[Kr] 833，上下文信息加密/解密单元 254 获得随机数 Kr．反之，从页表缓冲器 230 内取出对应于程序计数器（EIP）809 的执行码密钥 Kcode，并将它送到当前代码密钥存储单元 251．上下文信息加密/解密单元 254 利用执行码解密密钥 Kcode 将 E_{Kcode}[Kr]解密，并将结果送到执行码密钥与签名验证单元 257．执行码密钥与签名验证单元 257 验证 E_{Kcode}[Kr] 832 的解密结果是否与微处理器利用秘密密钥 Ks 的解密结果一致．通过验证，可以确认此上下文信息是在执行利用秘密密钥 Kcode 加密的执行码时产生的。

如果不对上下文信息进行执行码加密密钥验证，则用户可以通过产生利用任意适当秘密密钥 Ka 加密的代码进行攻击并将通过执行这些代码获得的上下文信息应用于利用其它秘密密钥 Kb 加密的代码。上述验证排除了进行这种攻击的可能性并保证了保护码上下文信息的安全。

通过将秘密执行码密钥 Kcode 附加到上下文信息，也可以实现此目的，但是在此实施例中，通过使用利用程序销售商选择的执行码密钥 Kcode 对其用于加密上下文信息的秘密随机数 Kr 进行加密的数值 E_{Kcode}[Kr]，可以降低保存上下文信息所要求的存储量，因此，可以实

现快速上下文转换和快速存储器保存．这还可以将上下文信息反馈到程序创建者．

当通过执行码密钥与签名验证单元 257 进行的执行码密钥的验证和签名的验证均成功时，将上下文恢复到寄存器文件 253，并且恢复程序计数器数值，结果，在执行产生此上下文的中断时，控制返回地址。

当这两种验证中的任何一个验证失败以致异常处理单元 255 引起异常产生时，异常出现地址指明发布转移指令或调用指令的地址．此外，将指明 TSS 非法的数值存储到 IDT 表内的中断起因字段，并将转移目标 TSS 地址存储到存储中断起因地址的寄存器．同样，OS 可以得知上下文转换失败的原因．

请注意，为了实现快速重新启动处理过程，还可以使用一种配置，在这种配置中，将由上下文信息加密/解密单元 254 加密的执行状态送到寄存器文件 253 与执行码密钥与签名验证单元 257 进行的验证处理过程并行进行，并且当验证失败时停止后续处理。

这种利用随机数的加密方法的安全性依赖于预测所使用的随机数序列的不可能性，在 Onodera 等人的第 2980576 号日本专利中披露了一种利用硬件产生非常难于预测的随机数的方法。

在根据用户使用程序的条件分析程序中产生故障的原因来改善程序质量方面，一个重要方面是程序销售商对上下文信息进行分析．在此实施例中，鉴于这种情况，所以采用既能实现上下文的安全性又能使程序销售商具有分析上下文信息的能力的上述方法，但是还有一个事实是使用该方法会增加上下文保存的开销．

此外，利用微处理器设定的签名对上下文信息进行验证可以防止利用任意选择的数值和密钥的组合执行非法上下文信息内的保护码，但是这种附加保护同样会增加开销。

因此，在不需要程序销售商具有分析上下文信息的能力或不需要排除利用非法上下文信息重新启动程序的机制的情况下，可以利用处理器的秘密密钥将含有用于标识执行码密钥信息的信息直接加密．即

使在这种情况下，仍可以使利用密码方法随意变更上下文成为不可能，并可以防止上下文信息被施加到利用不同密钥加密的程序。

这里将进一步说明上下文保存格式。之后说明其与运行的关系。

在图 10 中，“R”位 825－1 是指明上下文是否可以重新启动的位。当将此位设置为“1”时，通过利用上述恢复过程恢复保存在上下文中的状态可以重新启动执行，而当此位被设置为“0”时，不能进行重新启动。其效果是可以防止重新启动在加密程序的执行期间检测到其非法性的上下文，因此将可重新启动的上下文仅限制到处于适当状态的上下文。

“U”位 825－2 是指明 TSS 是用户 TSS 还是系统 TSS 的标志。当此位被设置为“0”时，则所保存的 TSS 是系统 TSS，而当此位被设置为“1”时，则所保存的 TSS 是用户 TSS。通过如上所述伴随改变异常入口特权的任务切换或通过任务门调用可以保存或恢复的 TSS 为系统 TSS.

系统 TSS 和用户 TSS 的差别在于，在恢复 TSS 时，指明当前执行程序的 TSS 保存地址的任务寄存器是否被更新。在系统 TSS 的恢复过程中，以链接到将被重新恢复的 TSS 的先前任务区域 801 的形式保存当前执行程序的任务寄存器，并将新 TSS 的段选择符读入任务寄存器。相反，在用户 TSS 的恢复过程中，不更新任务寄存器数值。用户 TSS 的目的仅在于程序寄存器状态的保存和恢复，因此特权模式不会发生变化。

异常包括应用程序进行系统调用的软件中断。在软件中断是为了进行系统调用的情况下，通用寄存器通常被用于参数交换，并且存在上下文信息加密过程妨碍参数交换的情况。

由于应用程序本身产生软件中断，所以在发生软件中断之前，应用程序可以破坏具有秘密的寄存器的信息。在这种假定条件下，可以使用一种只有软件中断情况不对寄存器进行加密的方法。当然，在这种情况下，应用程序创建者应考虑此事实并设计程序，从而可以保护程序的秘密。

01.02.14

接着将说明明文程序调试功能的抑制过程.

处理器具有: 分步执行功能, 每当执行指令时该功能总会产生中断; 和调试功能, 每当对特定地址进行存储器访问时该功能总会产生异常. 这些功能对于程序开发来说有用, 但是它们损害为了进行秘密保护被加密的程序的安全. 因此, 在此实施例的微处理器中, 在执行加密程序期间对这种调试功能进行了抑制.

指令 TLB 121 可以判别当前执行码是否被保护 (是否被加密). 在保护码的执行期间, 为了防止调试标志和调试寄存器侵入加密程序分析, 包括调试寄存器功能和分步执行功能的两种功能被禁止.

在调试寄存器功能中, 事先将作为执行码和执行数据的存储器访问范围和访问方式 (例如: 读/写) 设置到处理器内的调试寄存器, 这样, 每当发生相应的存储器访问时总会发生中断. 在此实施例中, 在保护码的执行期间, 设置到调试寄存器的内容将被忽略, 所以不会为了进行调试发生中断. 然而, 请注意, 调试位被设置到页表被排除在此规则外. 后面将对页表内的调试位进行说明.

在非保护码 (明文) 的执行中, 如果 EFLAGS 寄存器内的分步执行位被置位, 则每当执行一条指令时总会发生中断, 但是在保护码的执行期间, 此位也将被忽略, 因此不会发生中断.

在此实施例中, 除了为了防止分析加密执行码之外, 通过利用调试寄存器或调试标志防止对程序进行动态分析, 这些功能还使得用户难于对程序进行分析.

<数据保护>

接着将说明执行码处理目标数据的保护过程.

在此实施例中, 保护数据的加密属性被限定到在微处理器 101 内部设置的四个寄存器 CY0 至 CY3. 它们对应于图 9 所示的区域 717 至 720. 在图 9 中, 寄存器 CY0 至 CY2 的细节被省略, 只示出寄存器 CY3 的细节.

现在以 CY3 寄存器 717 为例说明加密属性的各元素. 在基地址 717 - 1 内规定指明待加密区域顶部的逻辑地址高位. 在大小区域 717 - 4

01.02.14

内规定区域的大小。由于区域的大小是以高速缓存行为单位说明的，所以在低位存在无效部分。在区域 717－5 规定数据加密密钥。由于这里使用了秘密密钥算法，所以还将区域 717－5 用于解密密钥。当规定密钥数值为“0”时，就意谓着该寄存器指明的区域未被加密。

在区域的说明中，CY0 给出最高优先权，而 CY1 至 CY3 依序给出连续较低优先权。例如，当 CY0 和 CY1 规定的区域重叠时，给出 CY0 的属性，即其优先权高于该区域内的 CY1 的优先权，此外，给出页表限定，即在作为执行码而不是作为处理目标数据进行存储器访问时具有最高优先权。

调试位 717－4 用于选择是在加密状态下还是在明文状态下进行调试状态中的数据运行。后面将说明调试位的细节。

图 12 示出在执行码处理目标数据的加密/解密过程中的信息流。这里，只在代码被保护状态下进行数据保护，即在加密状态下执行代码。然而，请注意，在以下说明的在调试状态下执行代码的情况被排除在此规则外。当代码被保护时，将数据加密控制寄存器（还可以称为：加密属性寄存器或数据保护属性寄存器）的内容 CY0 至 CY3 从图 4 所示的寄存器文件 253 读到在数据 TLB 141 内设置的数据密钥表 236。

当某些指令将数据写入逻辑地址“Addr”时，通过校验数据密钥表 236（参考图 4），数据 TLB 141 判别逻辑地址“Addr”是否包含在 CY0 至 CY3 的范围内。通过判别，如果规定加密属性，数据 TLB 141 命令代码加密功能块 212，在将对应的高速缓存行从 L1 数据高速缓存 218 存储器写入到存储器时利用规定的密钥对存储器内容加密。

同样，在读取情况下，如果目标地址具有加密属性，则数据 TLB 141 命令数据解密功能块 219，在将高速缓存行读出到相应的 L1 数据高速缓存 218 时利用规定的密钥将该数据解密。

在此实施例中，通过在微处理器 101 内部的寄存器内设置所有数据加密属性，防止数据加密属性被非法重写（包括 OS 特权），并在执行中断时将寄存器内容作为上下文信息以安全的方式保存到微处理器

101 外部的存储器（例如：图 4 所示的主存储器 281）.

正如上述关于上下文加密中说明的那样，以交错的高速缓存行为单位进行数据加密/解密。为此，即使在重写 L1 高速缓存 114 上的一个数据位时，则在存储器重写高速缓存行内的其它位。以高速缓存行为单位选择性地进行数据读/写，因此，增加的开销并不很大，但是，应该注意，不能以小于或等于高速缓存行大小的单位对加密存储区进行读/写。

以上说明了根据此实施例通过加密保护数据的方法。利用此方法，在主存储器上，可以通过利用密钥和应用程序规定的存储范围，在处理器内部将数据加密来处理加密数据，并根据应用程序的要求将它们作为明文数据读/写.

接着将说明两种机制，这两种机制均用于利用已读取这些数据的加密程序之外的程序（以下将被称为：其它程序）防止读取以明文形式存储在处理器内部的高速缓冲存储器内的数据.

首先，利用其密钥标识程序。利用在处理器内解密当前执行的指令时使用的密钥对象标识符进行此标识。在此，可以将密钥本身的数值用于此标识，但是执行码密钥的数值的大小多少有些大，即在解密前为 1024 位，而在解密后为 128 位。这样就会增加硬件的大小，因此使用总长度仅为 10 位的密钥对象标识符.

将存储解密执行码的 L1 指令高速缓存 213 具有与高速缓存行相符的属性存储器。当利用代码解密功能块 212 将解密执行码存储到 L1 指令高速缓存 213 时，将密钥对象标识符写入属性存储器.

此外，在从存储器读取加密数据并将它解密时，将数据保护属性寄存器 CY0 至 CY3 的内容从寄存器文件 253 读出到数据 TLB 141 的保护表管理功能块 233. 此时，还同时将对应于当前执行指令的密钥对象标识符从当前代码密钥存储单元 251 读出并保存到保护表管理功能块 233.

与指令高速缓存的情况相同，数据高速缓存 218 也具有与高速缓存行相符的属性存储器。当利用数据解密功能块 219 将从存储器读出

的数据解密并存储到 L1 数据高速缓存 218 时，将密钥对象标识符从保护表管理功能块 233 写入属性存储器。

当某些指令被执行并进行数据引用时，利用秘密保护破坏检测单元 256 将写入数据高速缓存属性的密钥对象标识符与指令高速缓存中该指令的密钥对象进行比较。如果它们不一致，则发生秘密保护破坏异常并且数据引用失败。在数据高速缓存属性指明明文的情况下，数据引用总是成功的。

请注意，当指令属性和数据属性不一致时，不是产生异常，而是会删除数据高速缓存的内容并再一次从存储器重读数据。

例如，现在研究执行码密钥以及数据保护属性寄存器 CY0 至 CY3 不同的程序－1 和程序－2。如果由程序－1 引用并写入高速缓存的加密数据被程序－2 引用，则程序－2 将读出不同的数据。此操作与保护秘密的目的一致。

如果两个程序具有相同数据密钥并且它们将相同地址的数据引用，则将读取相同数据，因此在它们之间可以共享此数据。

同样，在此实施例中，通过提供用于保持将执行指令的属性的功能块和指明它们最初所属的程序的数据，另一个程序－2 可以防止程序－1 产生的数据被引用，并在因为执行指令进行数据引用时，对它们的属性进行比较以检验它们是否一致。

<入口门>

在此实施例中，可以将控制从非保护码转移到保护码的情况仅限于下列两种:

（1） 与重新启动地址一致的、利用执行码密钥加密的上下文（即: 具有随机数的上下文）将被重新启动的情况; 以及

（2） 通过执行连续码或利用转移指令或调用指令，可以将控制从非保护码转移到保护码的入口门指令（“egate” 指令）.

设置这种限制是为了防止攻击者通过执行任意位置的代码获得代码分段信息。在关于上下文恢复的说明中已经说明了上述（1）的过程。即，只有当上下文信息与在含有中断之前立即执行的代码的执行码密

钥被验证一致，并验证微处理器 101 提供的正确签名被附加时，才将控制转移到保护码执行。

上述（2）是这样一种处理过程，即在将控制从非保护码转移到保护码的情况下，除非在控制开始时执行入口门（“egate”）指令调用的专用指令，否则就禁止转移到保护码的执行。

图 11 示出根据入口门指令转换保护域的过程。微处理器 101 将当前执行码的密钥保存到异常处理单元 131 的当前代码密钥存储单元 251（如图 4 所示）。首先，判别此密钥数值是否与指令的执行同时改变（步骤 601）。当检测到密钥数值发生变化时（步骤 601 中为 NO），接着就校验在密钥数值发生变化的同时执行的指令是否是入口门（“egate”）指令（步骤 602）。如果是入口门指令，就表示它是正确指令，因此可以将控制转移到已变化代码。因此，当被判别此指令是入口门指令（步骤 602 中为 YES）时，就执行此指令。

相反，当判别该指令不是入口门指令（步骤 602 中为 NO）时，就表示被中断的指令不是正确指令。既然如此，就判别刚执行的指令是否是加密（保护）指令（步骤 603）。如果是非保护指令，就直接进行异常处理，但是，如果是保护指令，则需要进行异常处理同时保护该指令。

因此，当判别该指令为非保护指令（步骤 603 中为 NO）时，就直接进行异常处理，而当判别该指令为保护指令（步骤 603 中为 YES）时，则进行不能重新启动的异常处理，同时保持该保护状态。

利用此控制转移限制，禁止将控制从明文码直接转移到非入口门指令地址处的代码。恢复上下文意谓着恢复曾经被该程序通过入口门执行的状态。因此，必须通过入口门才能执行保护程序。通过在程序中将放置入口门的地址抑制到最小必要数，产生的效果是，可以防止通过执行各种地址上的程序猜测程序结构来进行攻击。

此外，在此入口门，对此数据保护属性寄存器进行初始化。当执行此入口门时，将随机数 Kr 载入图 9 所示的数据保护属性寄存器 CY0 至 CY3 中的 717 至 720 内的密钥区域（CY3 内的区域 717－5）。将加

密目标顶部地址设置为“0”，将大小设置为存储器的最高限，并将逻辑地址空间设置为加密目标。如果在执行码中为设置调试属性，则将调试位（CY3 内的 717－3）设置为非调试。

换句话说，在开始执行加密码时，利用执行入口门时确定的随机数 Kr，将所有的存储器访问加密。此外，如上所述，在执行码加密控制中，在页表定义中给出更高优先权。所产生的此随机数 Kr 独立于在上下文加密中使用的随机数。

利用此机制，将被新执行的保护程序被设置为总是被利用在所有存储器访问开始时随机确定的密钥加密。

当然，在此状态下，整个存储区被加密，因此不可能通过存储器或与其它程序进行数据交换来调用系统参数。为此，通过置位数据保护属性寄存器连续调节其处理环境，以致将必要存储区转换到明文从而可以访问此存储区，该程序进行处理过程。通过在利用随机数加密的初始设置中给予寄存器 CY3 较低优先权，同时将密钥设置为“0”作为其它寄存器的明文访问设置，由于通过加密被保持秘密的明文数据和写入数据会因为错误被驱出明文区域，所以可以降低访问非必要区域的风险。

即使是在入口门进行初始化时，数据保护属性寄存器之外的寄存器的内容也不被加密，并且在此可以存储指明堆栈地址或参数地址的指针。然而，应该注意，在将通过入口门执行的程序的处理过程中，通过将非法数值设置到寄存器调用入口门，程序的秘密就不会被窃用。

还可以使用一种配置，在这种配置中，在认为安全具有更重要意义时，在入口门将包括除数据保护属性寄存器之外的通用寄存器在内的所有寄存器初始化，而不将标志和程序计数器初始化，但是这样会使编程更受限制并且效率更低。即使如此，也可以通过由程序计数器的相对地址或绝对地址指定的存储区交换诸如堆栈的参数。然而，请注意，与保存上下文的情况相同，包括部分标志寄存器和任务寄存器的系统寄存器被排除在加密寄存器或初始化寄存器的目标之外以便 OS 继续运行。

既然如此，在根据此实施例的微处理器中，由于在控制从明文状态转移到保护程序时，将将要执行的第一条指令限制为入口门指令，并且通过执行入口门指令将包括数据保护属性寄存器内在的寄存器初始化，所以可以防止分段执行保护码，尤其可以防止非法设置数据保护状态。

接着将说明保护程序的控制过程。首先说明在保护域内被关闭的调用和转移。保护域内中的调用与普通程序中的调用完全相同。图 13 概念性示出保护域内的调用和转移。

当保护域外的线程 1121 转移到保护域“egate”（入口门）指令时，保护域内的代码 1101 开始执行。通过执行“egate”指令，所有寄存器均被初始化，然后通过执行程序，顺序建立数据保护属性。利用“jmp xxx”指令（处理过程 1122），将控制转移到保护域内的转移目标“xxx” 1111，然后执行位于地址“ppp” 1112 的“call yyy”指令（处理过程 1123）。将调用源地址“ppp” 1112 推入堆栈存储器 1102，并将控制转移到调用目标“yyy” 1113。当调用目标处的处理被完成并执行“ret”指令时，将控制转移到堆栈返回地址“ppp” 1112。不存在对执行控制的限制，但是执行码密钥保持不变。

接着，将说明从保护域到非保护域的调用和转移过程。对于此控制转移，为了避免从保护域转移到程序创建者不希望的非保护域并为了保护数据保护状态，所以进行如下所述的专用指令的执行和用户 TSS 运行。

图 14 概念性示出从保护域转移到非保护域的调用和转移，其中保护域的执行码 1201 和非保护域的执行码 1202 被设置到各自域内。此外，提供用于与非保护域交换参数的用户 TSS 区域 1203 和区域 1204。

当线程 1221 执行“egate”指令时开始执行。在调用非保护域代码之前，保护域程序将用户 TSS 区域 1203 的地址保存到规定的参数区域 1204。然后，通过执行“ecall”指令调用非保护域代码。“ecall”指令有两个操作数，一个是调用目标地址，另一个是执行状态的保存目标。调用时，“ecall”指令将寄存器状态（或更准确地说，当程序计

数器处于发出“ecall”指令后的状态时的寄存器状态）以与上述描述的加密 TSS 情况的格式相同的格式保存到由操作数“uTSS”规定的区域。以下将此区域称为用户 TSS.

用户 TSS 与系统 TSS 之间的差别在于，图 10 所示的用户寄存器中，将 U 标志设置到 TSS 上的区域 825－2。后面将说明操作方面的差别。与将上下文信息保存到系统 TSS 的情况相同，在将用户 TSS 保存到存储器过程中，也未使用用户在数据保护属性寄存器 CY0 至 CY3 中规定的数据保护属性。

由于通过执行“ecall”指令寄存器被初始化，所以非保护域的调用目标代码不能交换参数。为此，从规定地址“param”1204 获得参数，并完成必要处理过程。在非保护域内对编程没有限制。在图 14 所示的实例中，调用子例程“qqq”1213（处理过程 1225）。通过设置用于将堆栈指针设置和参数拷贝到堆栈的适配器代码，从保护域的调用适用于子例程“qqq”的调用语义，例如在“exx”与“qqq”调用之间。通过存储器上的参数区域 1204 将处理结果送到调用源（处理过程 1226）。完成子例程的处理过程时，为了将控制返回调用源保护域，发出“sret”指令（处理过程 1227）。

与没有操作数的“ret”指令不同，“sret”指令具有一个用于规定用户 TSS 的操作数。在此，通过存储在参数区域“param”1204 的指针，可以间接将用户 TSS 1203 规定为恢复信息。利用“sret”指令的用户 TSS 恢复过程与系统 TSS 的恢复过程的主要差别在于，即使在恢复用户 TSS 时任务寄存器也完全不会受影响。用户 TSS 的任务链接字段被忽略。当在“sret”指令的操作数中规定具有被设置为“0”的 U 标志的系统 TSS 时，恢复将失败。

进行恢复时，进行上述说明的执行状态的解密过程和执行码密钥与签名的验证过程，并且当检测到破坏时，将产生秘密保护破坏异常。当验证成功时，从紧跟在调用源“ecall”指令之后的指令开始重新启动执行。此地址被加密并被标记到用户 TSS，因此用密码的方法不可能伪造此地址。除程序计数器之外的所有寄存器被设置到调用前的状

态，保护域的代码从参数区域 1204 获得子例程“exx”的执行结果。

在完成保护域的处理过程后将控制转移到非保护域时，使用“ejmp”指令。与“ecall”指令不同，“ejmp”指令不进行状态保存。如果利用“ecall”指令和“ejmp”指令之外的指令（例如：“jmp”指令和“call”指令）将控制从保护域转移到非保护域，则产生秘密保护破坏异常并将加密上下文信息保存到系统 TSS 区域（任务寄存器指明的区域）。请注意，此时，将上下文信息标记为不能重新启动。还请注意，将保护域内的地址规定为“ejmp”指令的转移目标不会产生破坏。

现在完成了对从保护域到非保护域的调用过程以及在此过程中新增加的指令的说明。

应用程序恢复用户 TSS 时，利用具有特权的 OS 替换用户 TSS 进行攻击不是完全不可能。然而，只要正确管理保护域执行码密钥，在这种情况下可交换的 TSS 信息显然只有上下文信息，上下文信息的执行总是通过“egate”指令开始进行，并且通过保存由中断或用户产生的执行状态来保存上下文信息。由于交换此上下文信息而泄露应用程序的秘密的可能性非常小，并且攻击者非常难于猜测到在获取应用程序秘密时哪种上下文信息交换是必须的。

如果在调用目标首先执行的是调用源侧的“egate”指令，则上述描述的从保护域到非保护域的调用过程还可以应用于控制在保护域之间转移的过程。

既然如此，通过对在保护域之间交换参数的区域进行加密，通过利用共享的密钥事先在这些保护域之间进行验证密钥交换，可以在保护域之间进行安全调用。

如上所述，根据本方面的微处理器，在多任务情况下，通过利用加密过程保护执行码和执行码的处理目标数据，可以防止 OS 或第三者进行非法分析。

此外，在保存加密数据情况下，它还可以防止非法重写加密属性。

此外，利用任意随机数 Kr 而不是利用固定密钥作为处理目标数据的密钥，它还可以防止加密数据被非法攻击。

01-02-14

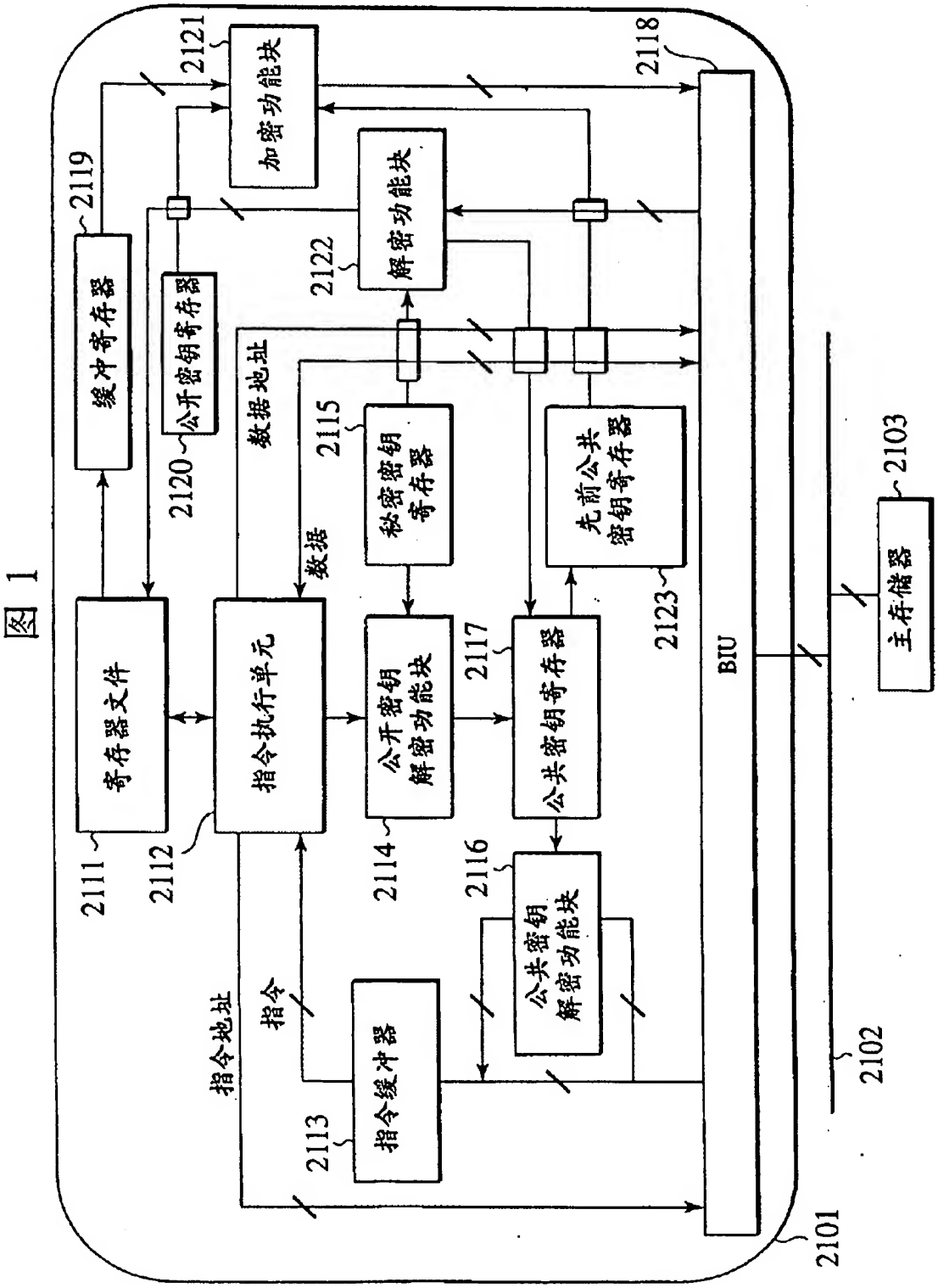
此外，它还可以在明文状态进行调试，并且当发现错误时，可以将错误反馈送到知道执行码密钥的程序销售商。

此外，通过附加微处理器的签名将要求秘密保护的信息（例如：加密属性信息）保存到外部存储器，通过仅将必要部分读入微处理器内部的寄存器，以及通过在读取时进行签名验证，它还可以防止增加微处理器内的存储器并降低微处理器的成本。在此方法中，还可以确保防止读取时进行置换。

还请注意，除了以上说明的方法之外，根据上述实施例所作的许多变更和变换均离不开本发明的新颖特征和优势特征。因此，所有这些变更或变换均应属于所附权利要求所述的本发明范围。

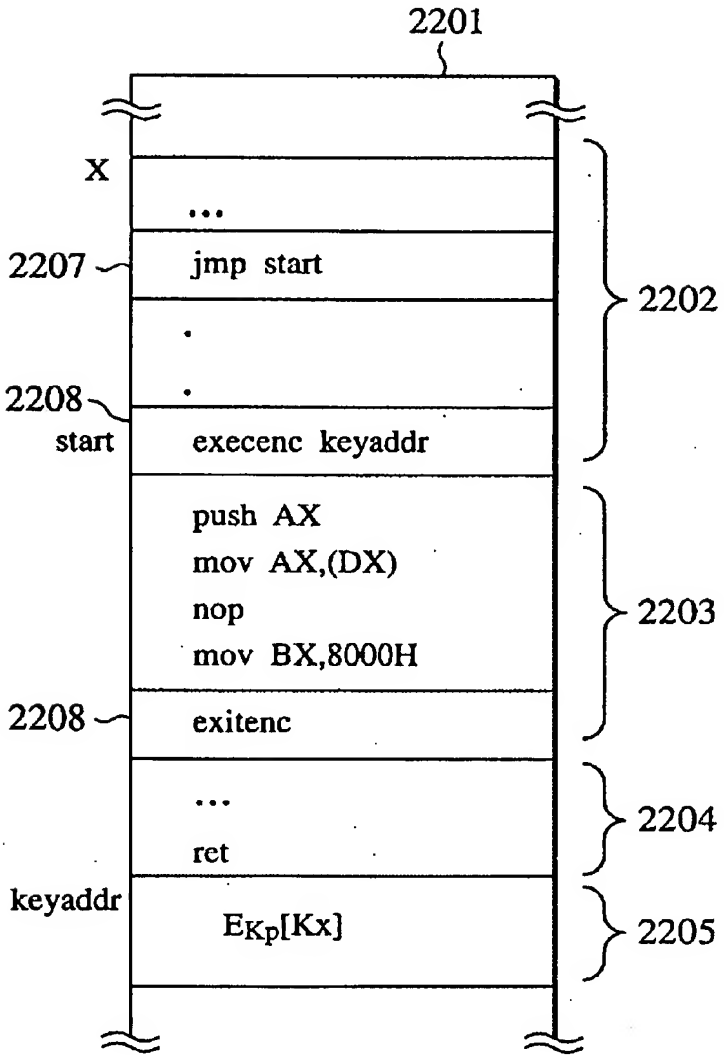
说 明 书 附 图

图 1


2101
2102
2103
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
寄存器文件
指令执行单元
指令缓冲器
公开密钥解密功能块
秘密密钥寄存器
公共密钥解密功能块
公共密钥寄存器
先前公共密钥寄存器
缓冲寄存器
公开密钥寄存器
加密功能块
解密功能块
BIU
主存储器
指令地址
指令
数据地址
数据

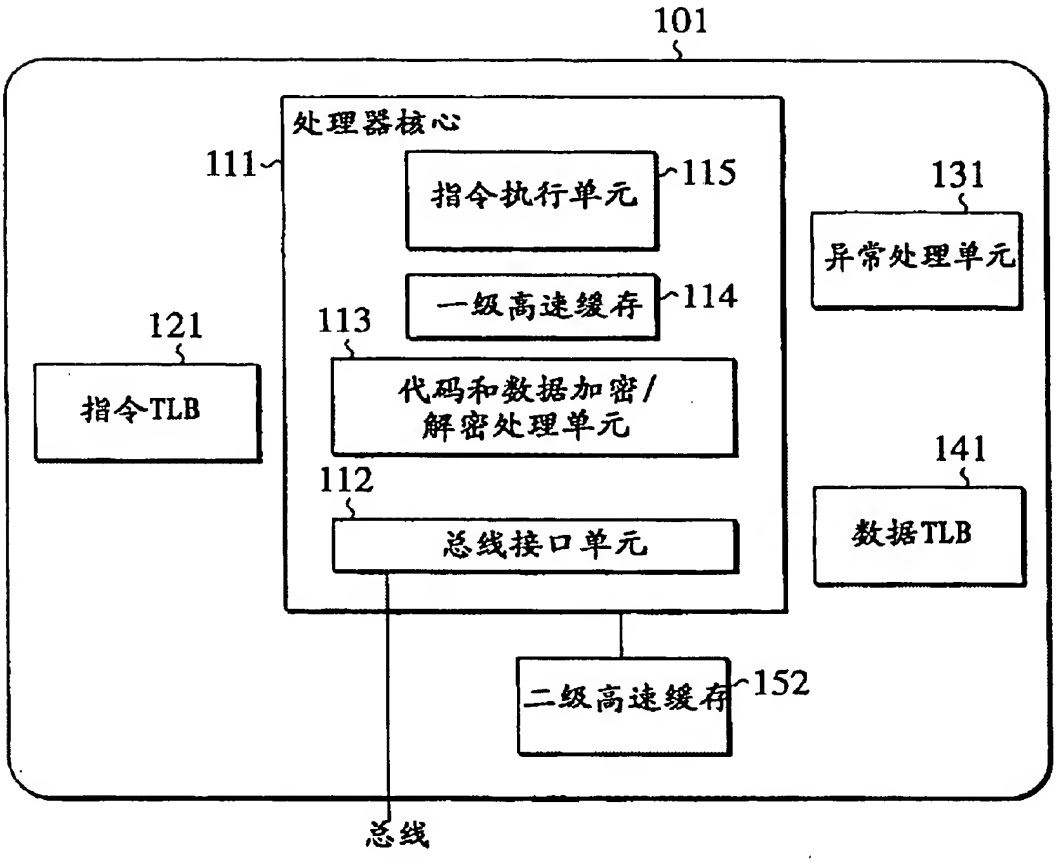
01.02.14

图 2


2201
X
...
2207
jmp start
2202
2208
start
execenc keyaddr
push AX
mov AX,(DX)
nop
mov BX,8000H
2203
2208
exitenc
...
ret
2204
keyaddr
EKp[Kx]
2205

01.02.14

图 3


101
处理器核心
111
指令执行单元
115
131
异常处理单元
一级高速缓存
114
121
113
指令TLB
代码和数据加密/
解密处理单元
141
112
总线接口单元
数据TLB
二级高速缓存
152
总线

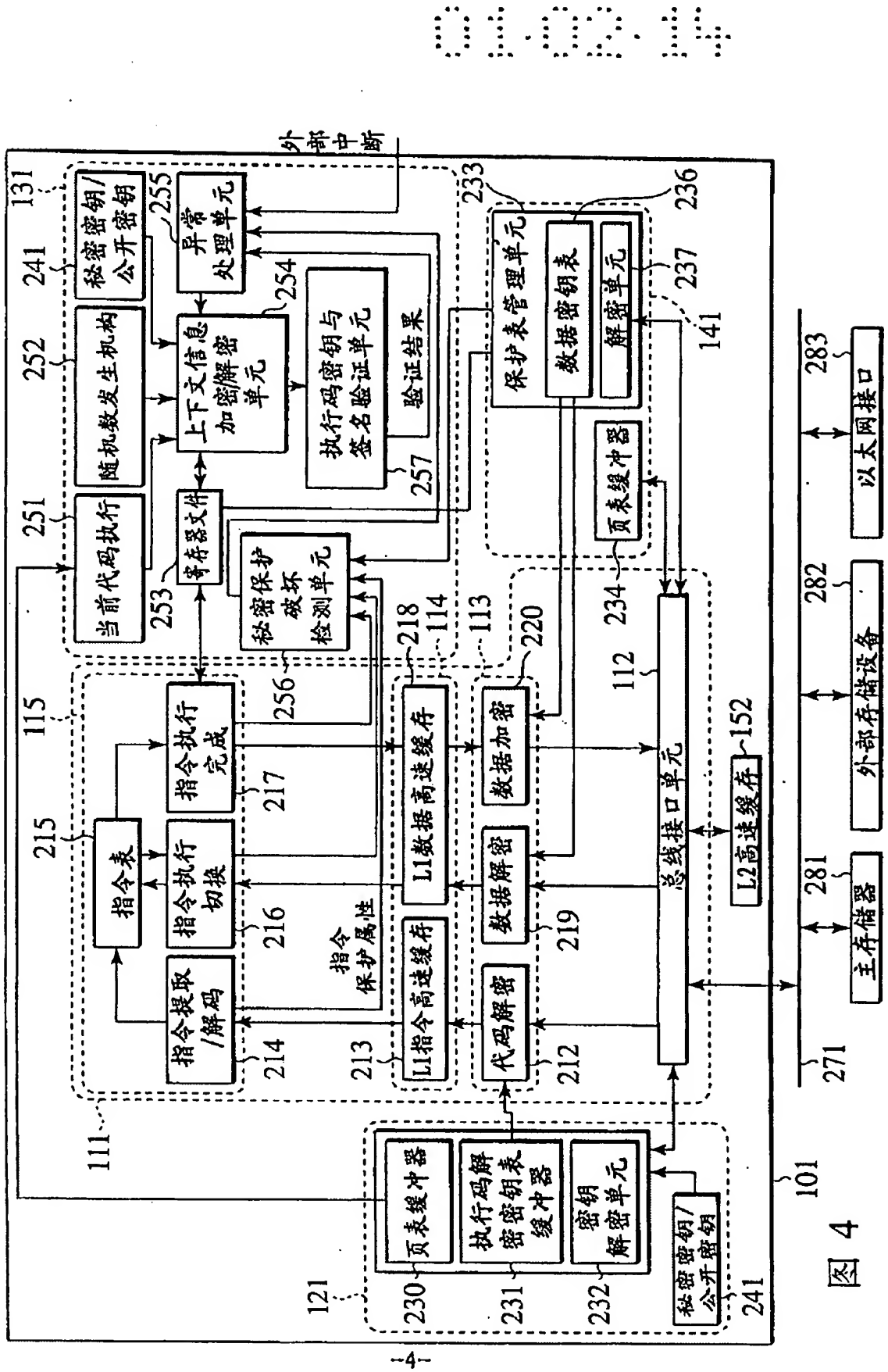
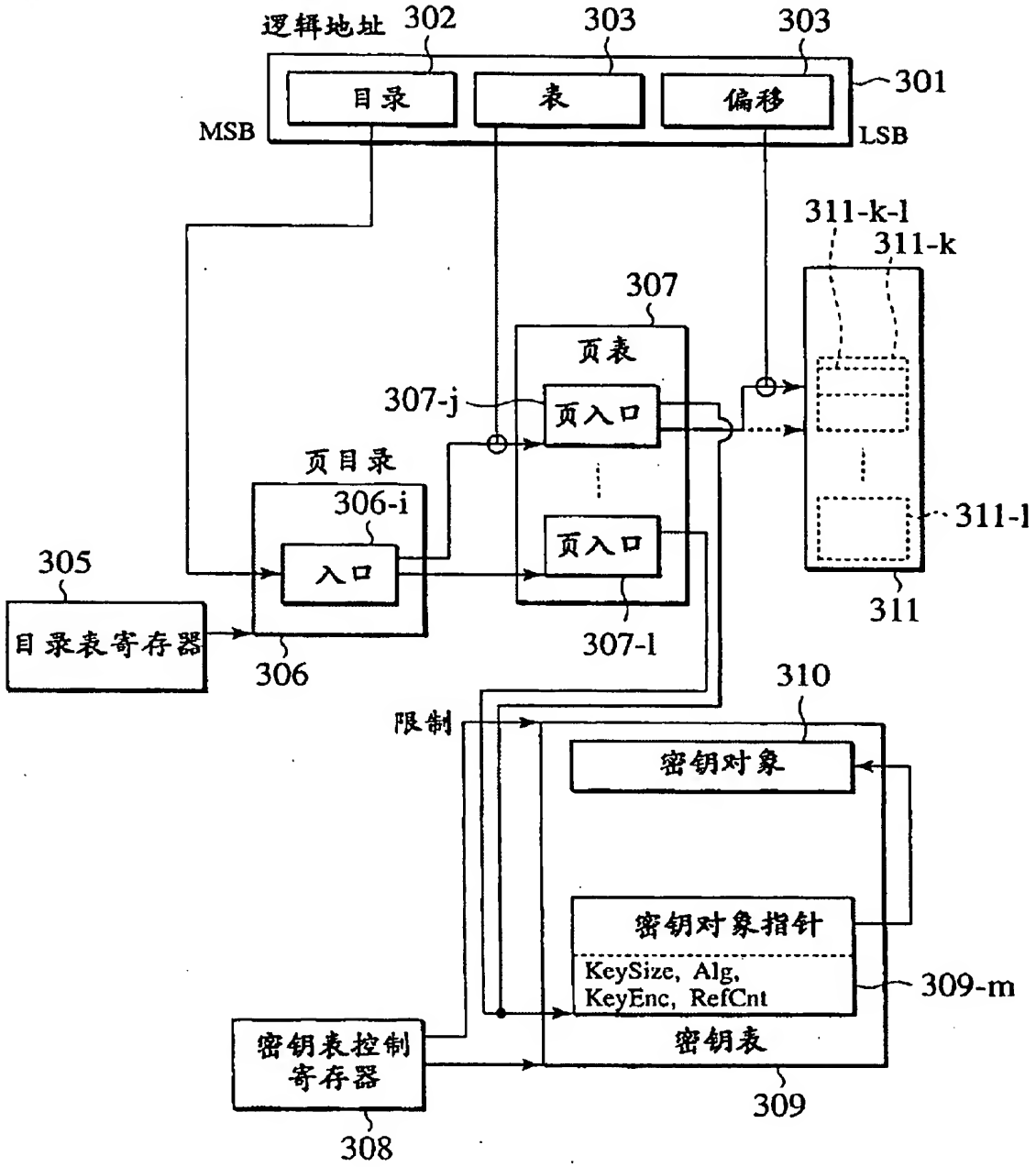

外部中断
131
241
252
251
255
254
253
233
236
237
141
283
282
281
271
101
121
111
115
215
216
217
256
218
114
113
220
234
112
152
213
214
212
219
230
231
232
241
秘密密钥/公开密钥
随机数发生机构
当前代码执行
异常处理单元
上下文信息加密/解密单元
寄存器文件
执行码密钥与签名验证单元
验证结果
保护表管理单元
数据密钥表
解密单元
页表缓冲器
秘密保护破坏检测单元
指令执行完成
指令表
指令执行切换
指令提取/解码
指令保护属性
L1数据高速缓存
L1指令高速缓存
数据加密
数据解密
代码解密
总线接口单元
L2高速缓存
以太网接口
外部存储设备
主存储器
页表缓冲器
执行码解密密钥表缓冲器
密钥解密单元
秘密密钥/公开密钥

图 4

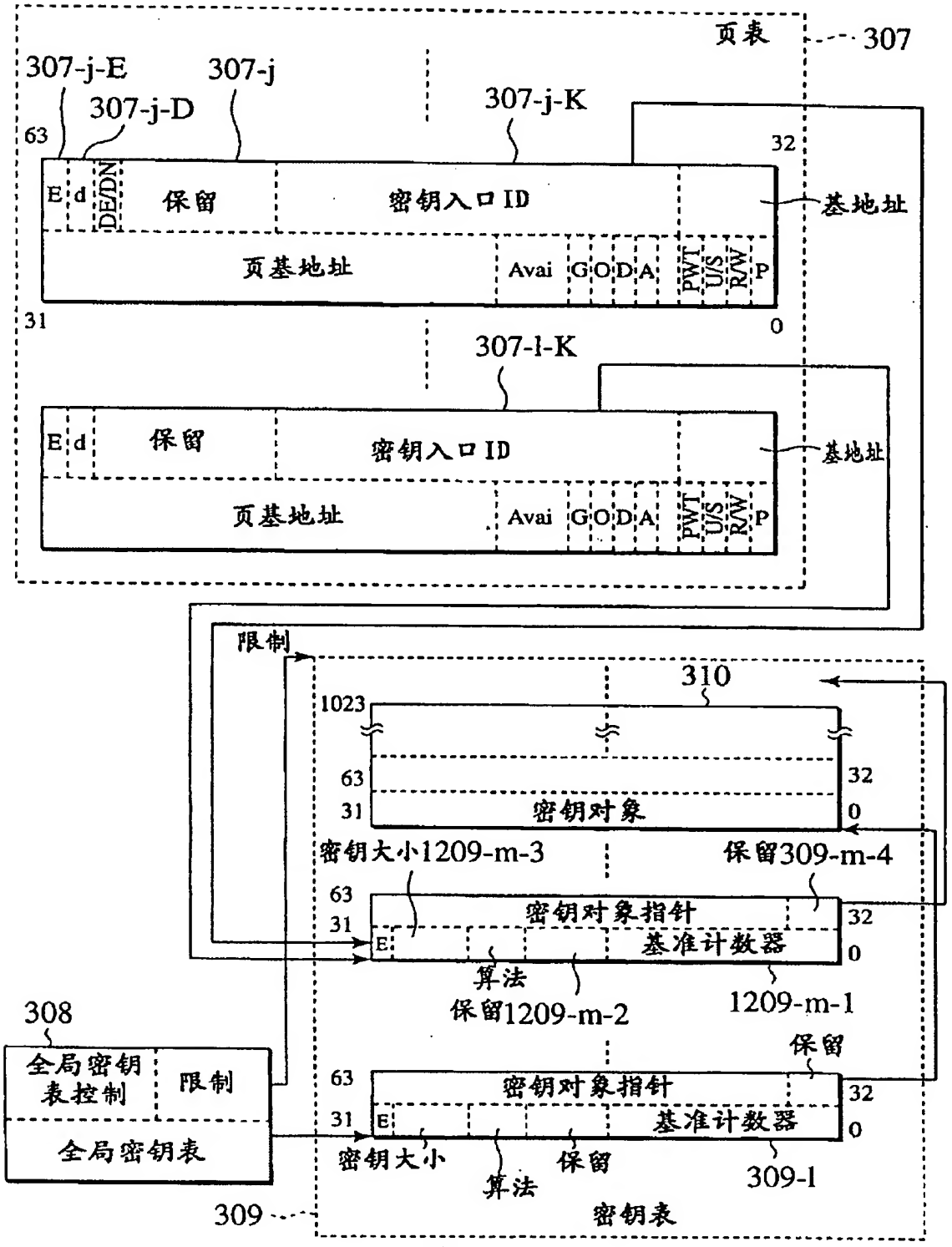
01.02.14

图 5、


逻辑地址
302
303
303
301
目录
表
偏移
MSB
LSB
311-k-1
311-k
307
页表
307-j
页入口
页目录
306-i
入口
页入口
311-1
305
目录表寄存器
306
307-1
311
310
限制
密钥对象
密钥对象指针
KeySize, Alg,
KeyEnc, RefCnt
309-m
密钥表控制
寄存器
密钥表
309
308

01.02.14

图 6


页表
307
307-j-E
307-j
307-j-D
307-j-K
63
32
E
d
DE/DN
保留
密钥入口 ID
基地址
页基地址
Avai
G
O
D
A
PWT
U/S
R/W
P
31
0
307-l-K
限制
310
1023
密钥对象
密钥大小1209-m-3
保留309-m-4
密钥对象指针
基准计数器
算法
保留1209-m-2
1209-m-1
308
全局密钥表控制
全局密钥表
保留
密钥大小
309-1
309
密钥表

01.02.14

图7A

	0	1	2	3
0	A0	A1	A2	A3
1	B0	B1	B2	B3
2	C0	C1	C2	C3
3	D0	D1	D2	D3
4	E0	E1	E2	E3
5	F0	F1	F2	F3
6	G0	G1	G2	G3
7	H0	H1	H2	H3

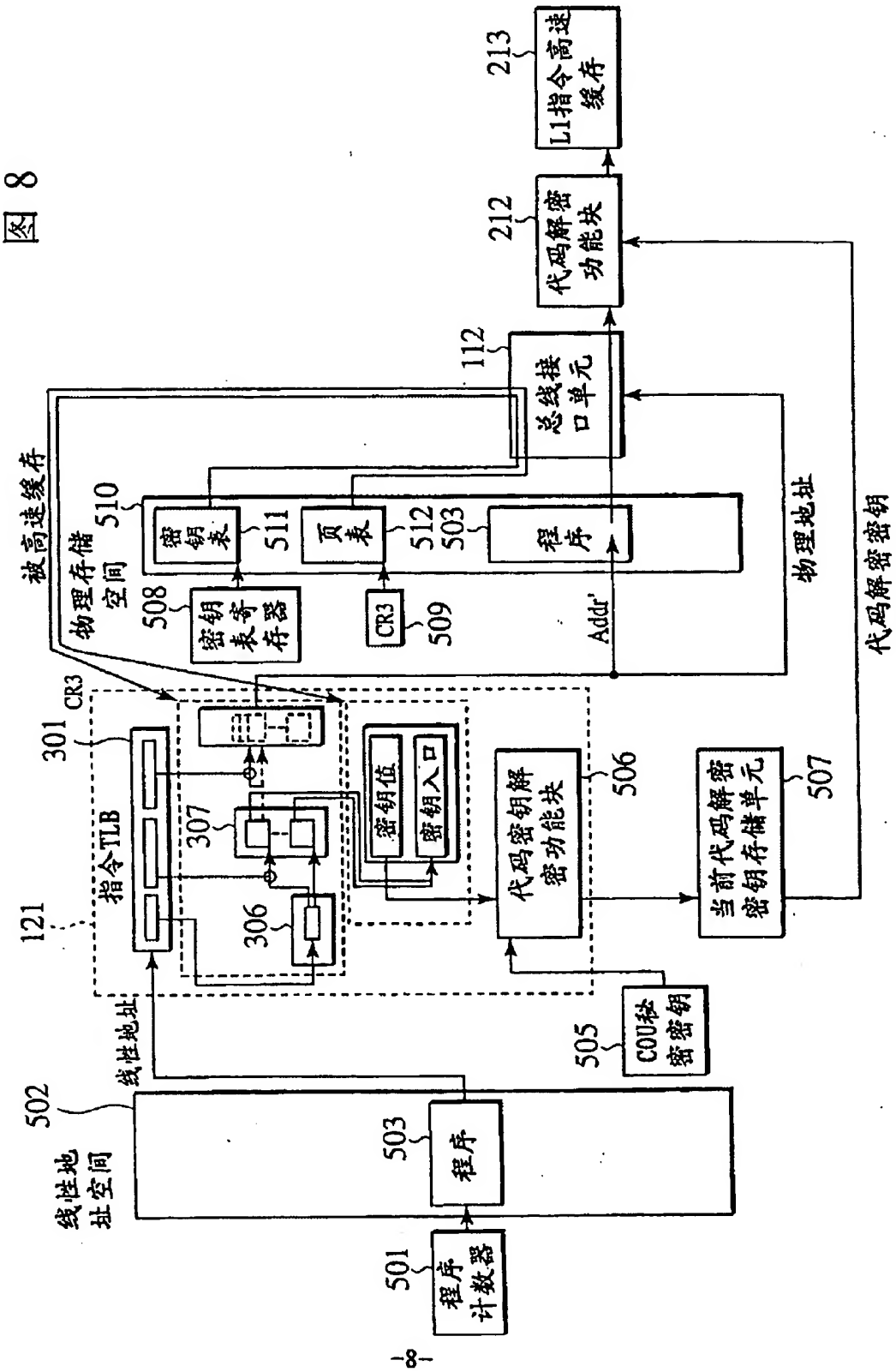
交错前

图7B

0	A0	B0	C0	D0
1	E0	F0	G0	H0
2	A1	B1	C1	D1
3	E1	F1	G1	H1
4	A2	B2	C2	D2
5	E2	F2	G2	H2
6	A3	B3	C3	D3
7	E3	F3	G3	H3

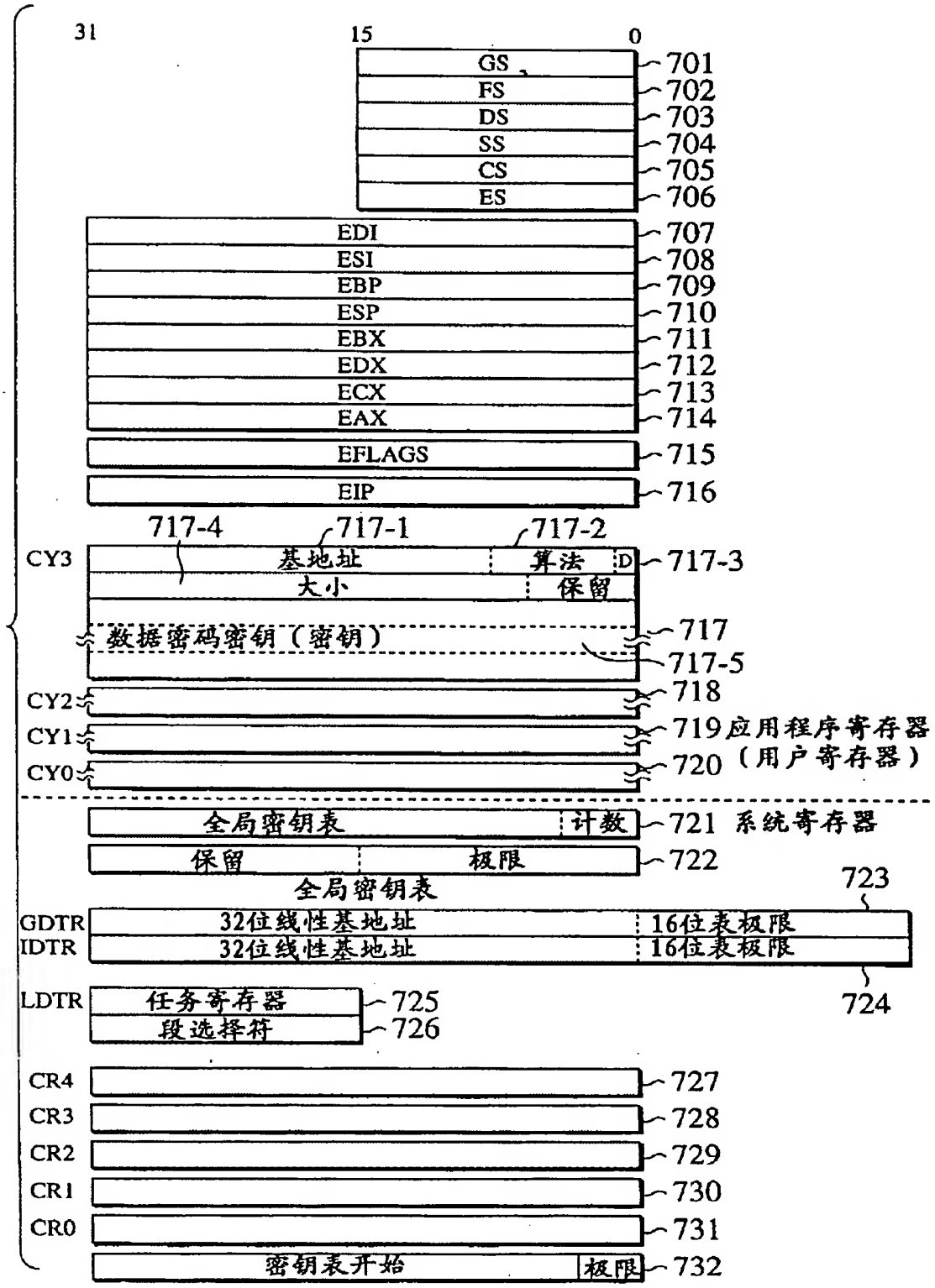
交错后

图 8


213
L1指令高速缓存
212
代码解密功能块
112
总线接口单元
被高速缓存
物理存储空间
510
密钥表
511
页表
512
503
程序
508
密钥表寄存器
CR3
509
Addr'
物理地址
代码解密密钥
301
CR3
指令TLB
121
307
306
密钥位
密钥入口
代码密钥解密功能块
506
当前代码解密密钥存储单元
507
505
COU私密密钥
线性地址
502
线性地址空间
503
程序
501
程序计数器

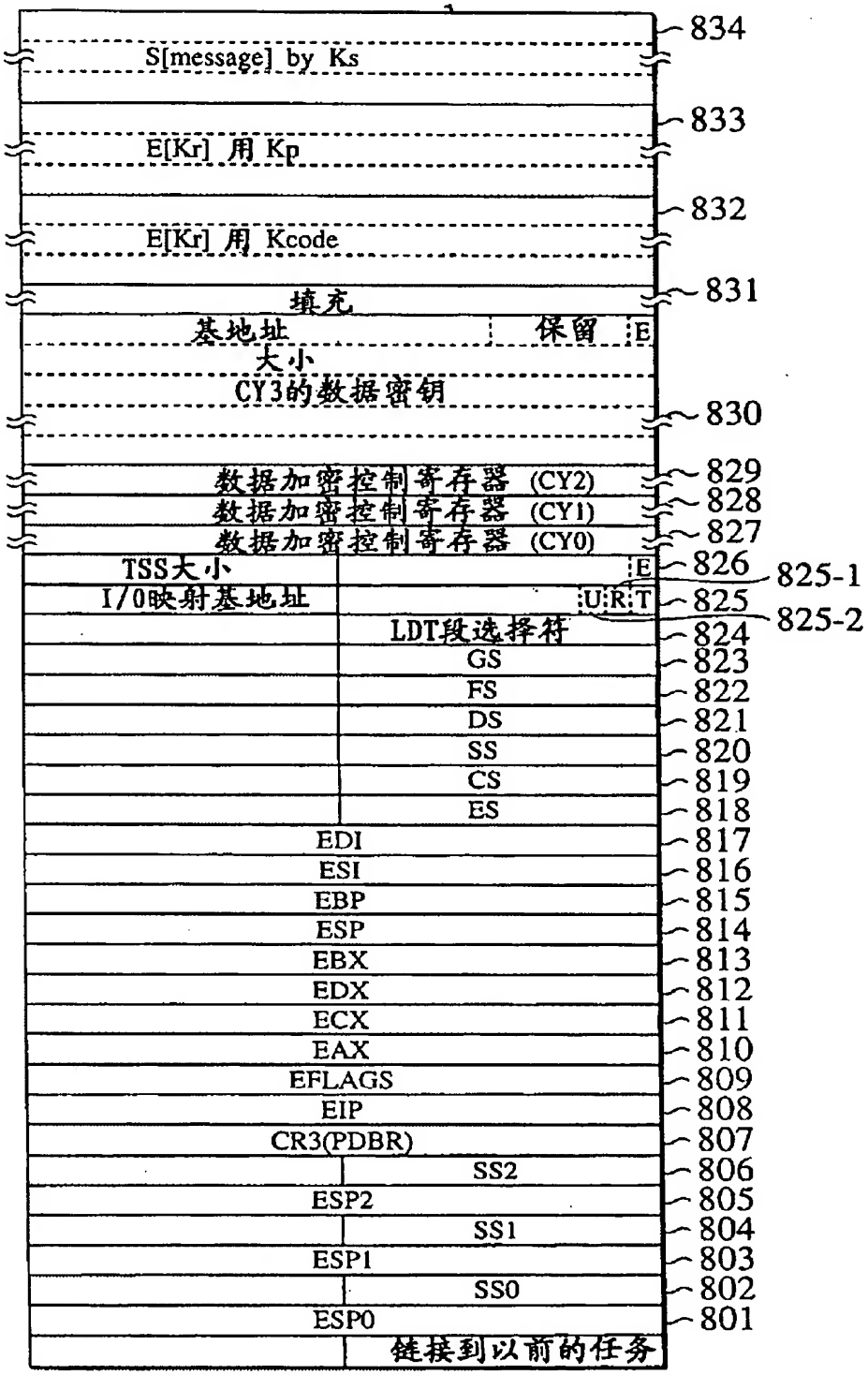
01.02.14

图 9


31
15
0
GS
FS
DS
SS
CS
ES
701
702
703
704
705
706
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
707
708
709
710
711
712
713
714
EFLAGS
715
EIP
716
717-4
717-1
717-2
CY3
基地址
算法
D
717-3
大小
保留
数据密码密钥（密钥）
717
717-5
CY2
718
CY1
719 应用程序寄存器
CY0
720 （用户寄存器）
全局密钥表
计数
721 系统寄存器
保留
极限
722
全局密钥表
723
GDTR
32位线性基地址
16位表极限
IDTR
32位线性基地址
16位表极限
724
LDTR
任务寄存器
725
段选择符
726
CR4
727
CR3
728
CR2
729
CR1
730
CR0
731
密钥表开始
极限
732

01.02.14

图 10


834
S[message] by Ks
833
E[Kr] 用 Kp
832
E[Kr] 用 Kcode
831
填充
基地址
保留
E
大小
CY3的数据密钥
830
数据加密控制寄存器 (CY2)
829
数据加密控制寄存器 (CY1)
828
数据加密控制寄存器 (CY0)
827
TSS大小
E
826
825-1
I/O映射基地址
U R T
825
825-2
LDT段选择符
824
GS
823
FS
822
DS
821
SS
820
CS
819
ES
818
EDI
817
ESI
816
EBP
815
ESP
814
EBX
813
EDX
812
ECX
811
EAX
810
EFLAGS
809
EIP
808
CR3(PDBR)
807
SS2
806
ESP2
805
SS1
804
ESP1
803
SS0
802
ESP0
801
链接到以前的任务

01.02.14

图 11

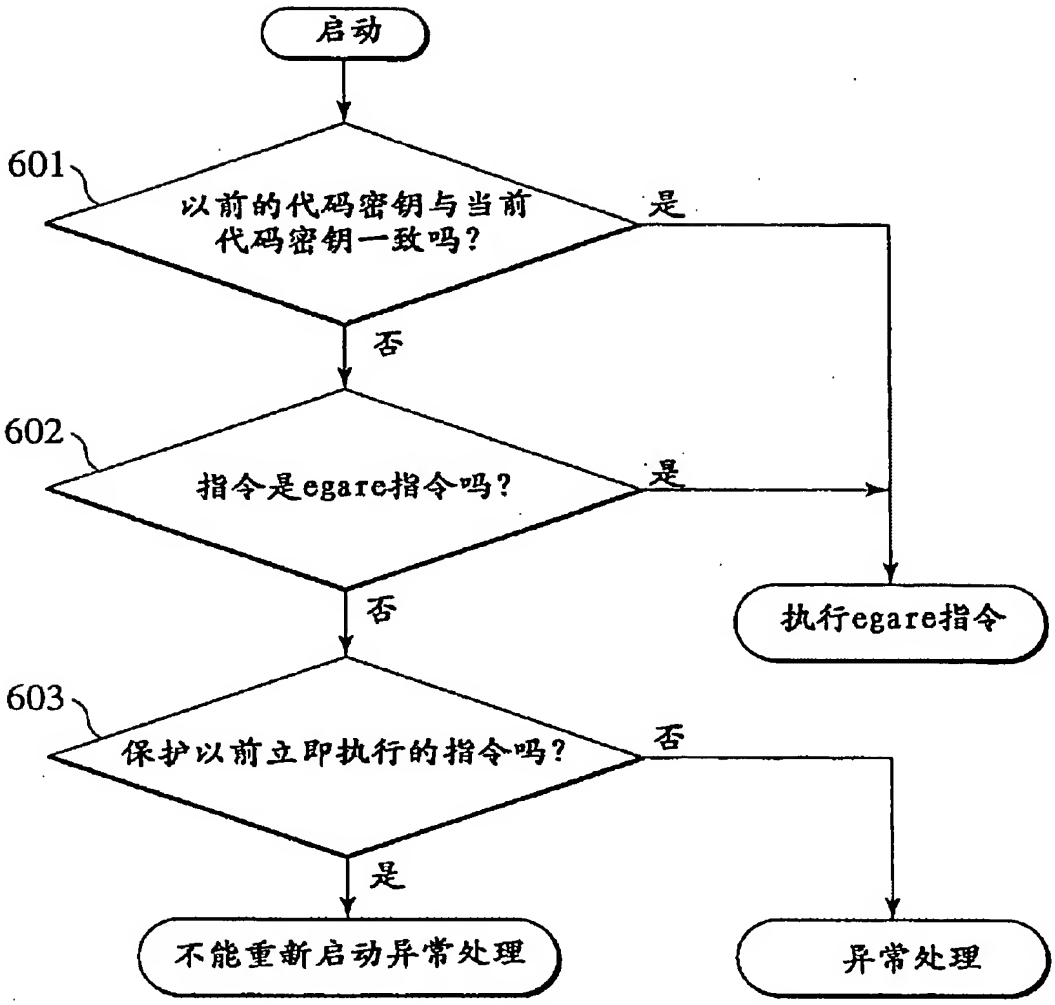
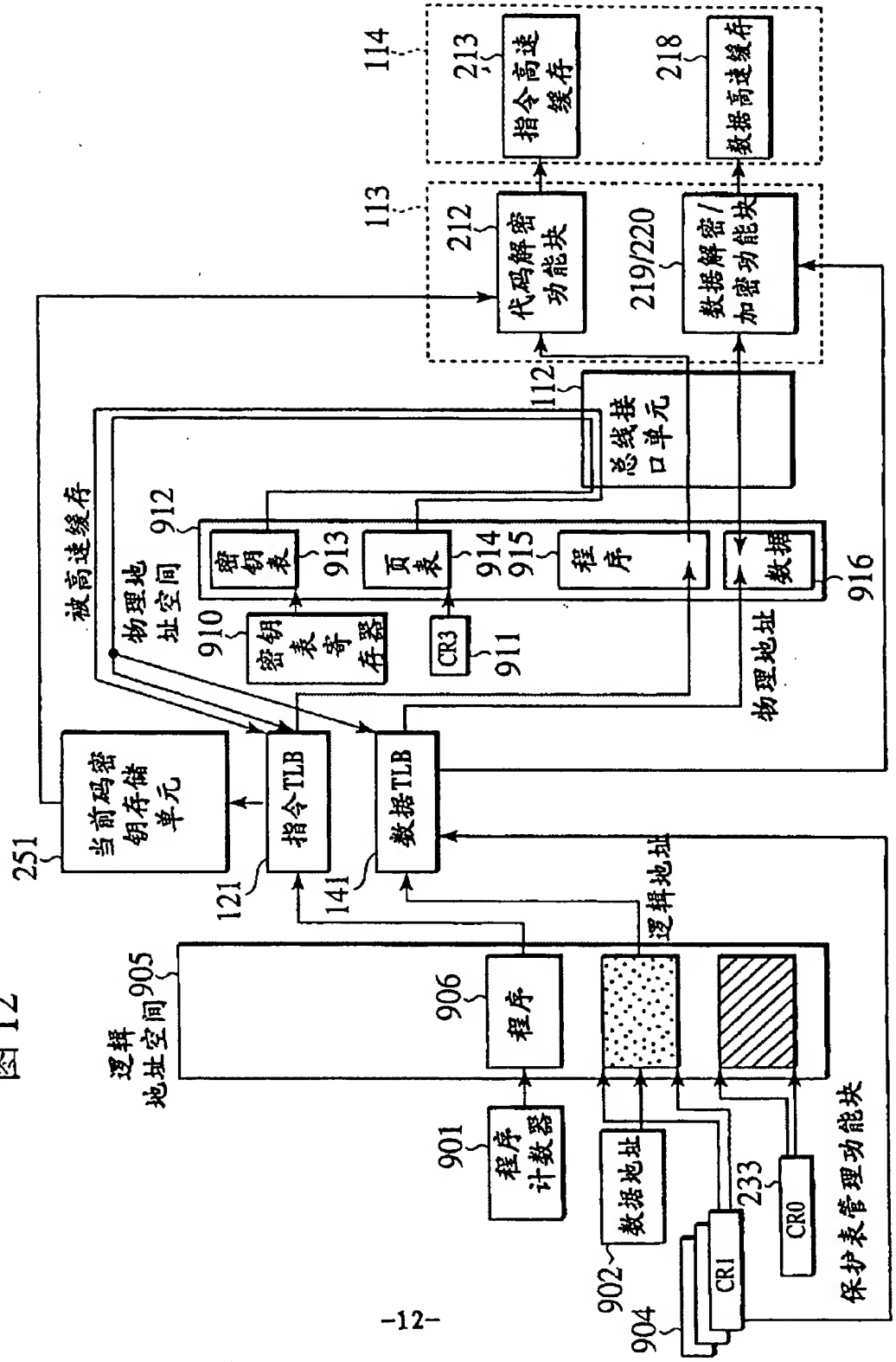
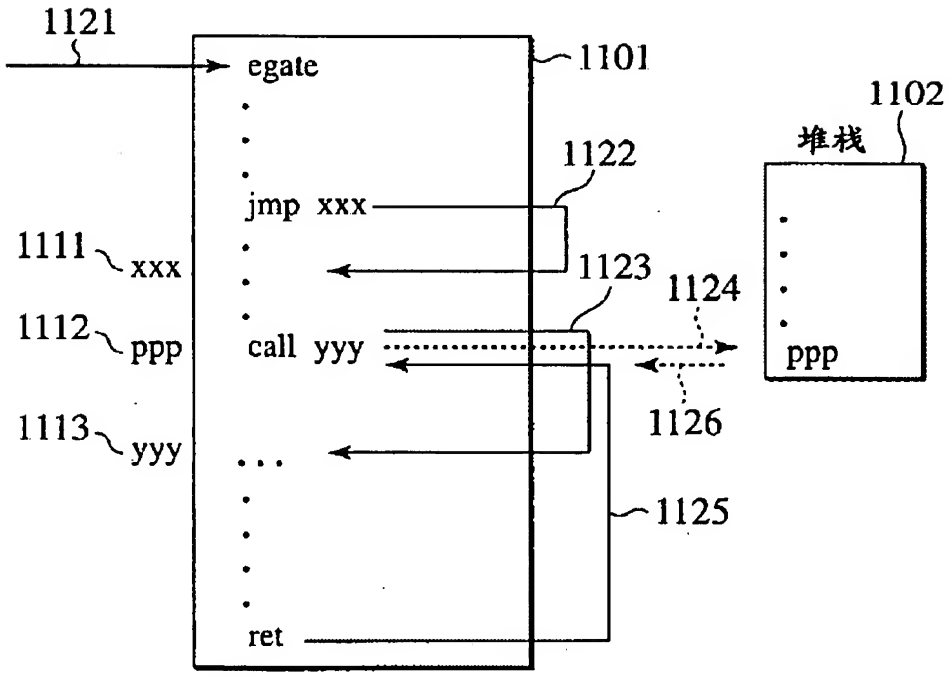

启动
601
以前的代码密钥与当前代码密钥一致吗?
是
否
602
指令是egare指令吗?
是
否
执行egare指令
603
保护以前立即执行的指令吗?
否
是
不能重新启动异常处理
异常处理

图 12


逻辑地址空间 905
程序 906
程序计数器 901
数据地址 902
CR1 904
CR0 233
保护表管理功能块
逻辑地址
指令TLB 121
数据TLB 141
当前码密钥存储单元 251
物理地址
被高速缓存
物理地址空间 912
密钥表 913
页表 914
915
程序
数据 916
密钥表寄存器 910
CR3 911
总线接口单元 112
113
代码解密功能块 212
数据解密/加密功能块 219/220
114
指令高速缓存 213
数据高速缓存 218

01.02.14

图 13


1121
egate
1101
1102
堆栈
1122
jmp xxx
1111
xxx
1123
1124
1112
ppp
call yyy
ppp
1126
1113
yyy
...
1125
ret

01.02.14

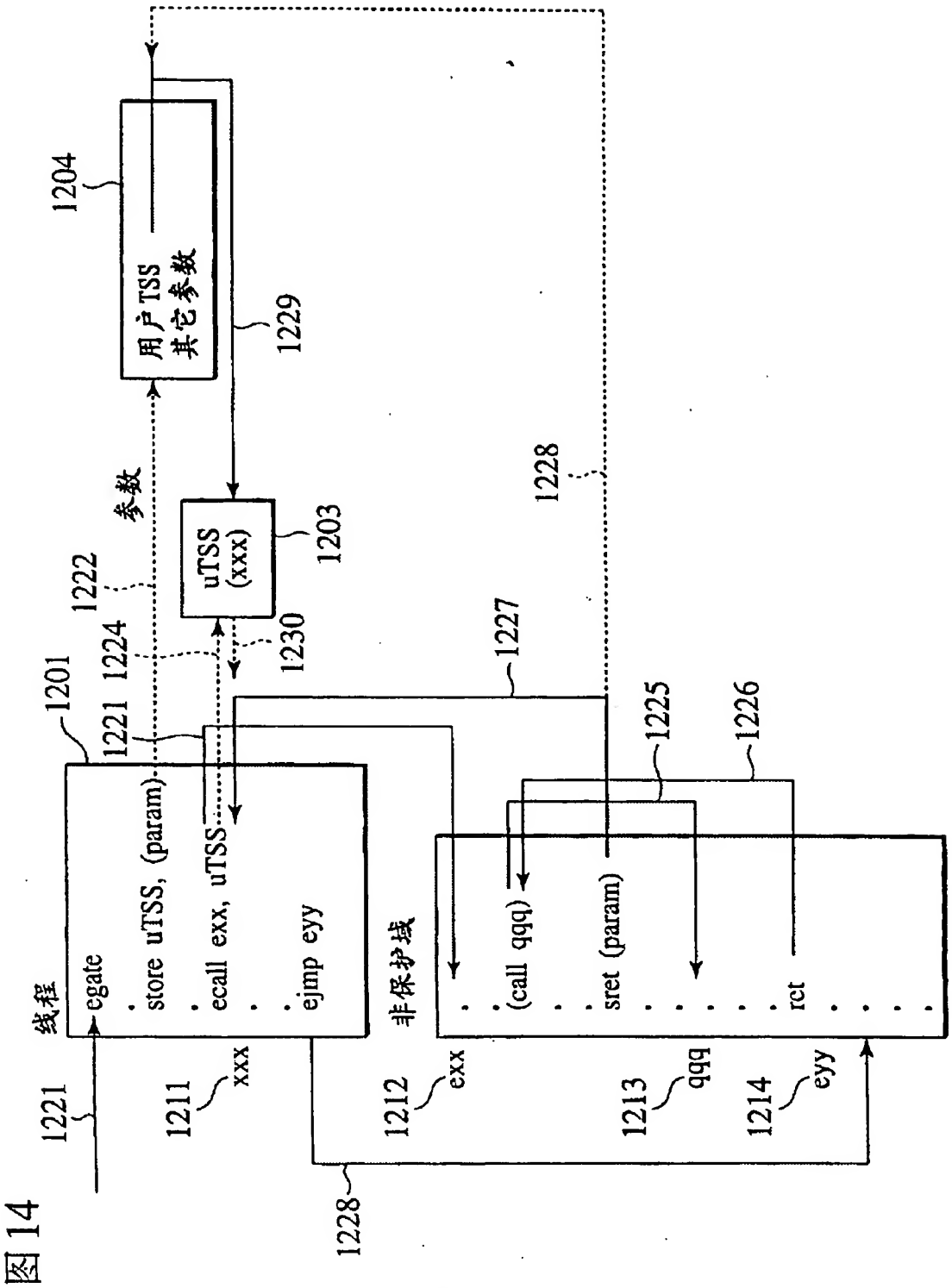

1204
用户TSS
其它参数
1229
参数
1222
uTSS
(xxx)
1203
1228
1224
1230
1227
1201
1221
1225
1226
store uTSS, (param)
ecall exx, uTSS
ejmp eyy
egate
线程
非保护域
(call qqq)
sret (param)
rct
1221
1211
xxx
1212
exx
1213
qqq
1214
eyy
1228

图 14

01-02-14

图 15

31 15	0	
I/O映射基地址	LDR段选择符 T	100
		96
	GS	92
	FS	88
	DS	84
	SS	80
	CS	76
	ES	72
EDI		68
ESI		64
EBP		60
ESP		56
EBX		52
EDX		48
ECX		44
EAX		40
EFLAGS		36
EIP		32
CR3(PDBR)		28
	SS2	24
ESP2		20
	SS1	16
ESP1		12
	SS0	8
ESP0		4
	链接到以前的任务	0

US006983374B2

(12) **United States Patent**
Hashimoto et al.

(10) **Patent No.: US 6,983,374 B2**
(45) **Date of Patent: Jan. 3, 2006**

(54) **TAMPER RESISTANT MICROPROCESSOR**

(75) Inventors: **Mikio Hashimoto**, Chiba (JP); **Keiichi Teramoto**, Tokyo (JP); **Takeshi Saito**, Tokyo (JP); **Kenji Shirakawa**, Kanagawa (JP); **Kensaku Fujimoto**, Kanagawa (JP)

(73) Assignee: **Kabushiki Kaisha Toshiba**, Kawasaki (JP)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 759 days.

(21) Appl. No.: **09/781,158**

(22) Filed: **Feb. 13, 2001**

(65) **Prior Publication Data**

US 2001/0018736 A1 Aug. 30, 2001

(30) **Foreign Application Priority Data**

Feb. 14, 2000 (JP) ..... P 2000-035898
May 8, 2000 (JP) ..... P 2000-135010

(51) **Int. Cl.**
G06F 12/14 (2006.01)

(52) **U.S. Cl.** **713/194**; 713/189

(58) **Field of Classification Search** 713/194, 713/189, 190, 200, 193
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,558,176	A	* 12/1985	Arnold et al.	713/190
4,634,807	A	* 1/1987	Chorley et al.	705/55
4,757,533	A	* 7/1988	Allen et al.	713/192
4,847,902	A	7/1989	Hampson	
5,123,045	A	6/1992	Ostrovsky et al.	
5,224,166	A	6/1993	Hartman, Jr.	
5,548,645	A	8/1996	Ananda	
5,666,411	A	9/1997	McCarty	

(Continued)

FOREIGN PATENT DOCUMENTS

EP	0 583 140	2/1994
GB	2 330 932	5/1999
JP	05-020197	1/1993
JP	06-112937	4/1994
JP	08-305558	11/1996
JP	2980576	9/1999
JP	11-282667	10/1999

OTHER PUBLICATIONS

"The TrustNo 1 Cryptoprocessor Concept"□□Markus Kuhn□□Apr. 30, 1997.*

Bruce Schneier, Applied Cryptography, Second Edition, 1996.*

Markus Kuhn, "The Trust No. 1 Cryptoprocessor Concept," CS555 Report, Purdue University, Apr. 30, 1997, pp. 1–6.

Tanguy Gilmont, et al., "an Architecture of Security Management Unit for Safe Hosting of Multiple Agents", Microelectrics Laboratory, Université Catholique de Louvain, pp. 1–12.

Tanguy Gilmont, et al., "Enhancing Security in the Memory Management Unit", Proceedings of 25th EUROMICRO Conference, Sep. 1999, pp. 1–8.

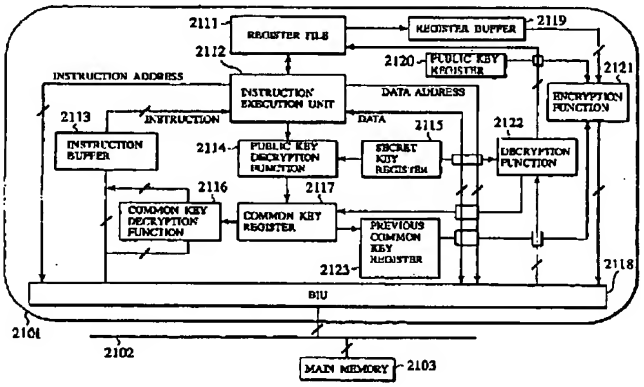
(Continued)

Primary Examiner—Gregory Morse
Assistant Examiner—Andrew Nalven
(74) *Attorney, Agent, or Firm*—Oblon, Spivak, McClelland, Maier & Neustadt, P.C.

(57) **ABSTRACT**

Under a multi-task environment, a tamper resistant microprocessor saves a context information for one program whose execution is to be interrupted, where the context information contains information indicating an execution state of that one program and the execution code encryption key of that one program. An execution of that one program can be restarted by recovering the execution state of that one program from the saved context information. The context information can be encrypted by using the public key of the microprocessor, and then decrypted by using the secret key of the microprocessor.

2 Claims, 15 Drawing Sheets



U.S. PATENT DOCUMENTS

5,805,706	A		9/1998	Davis
5,825,878	A		10/1998	Takahashi et al.
5,894,516	A		4/1999	Brandenburg
6,003,117	A		12/1999	Buer et al.
6,006,328	A	*	12/1999	Drake 713/200
6,385,727	B1	*	5/2002	Cassagnol et al. 713/193
6,651,171	B1	*	11/2003	England et al. 713/193

OTHER PUBLICATIONS

Larry W. Allen, et al., Usenix, pp. 145–160, "Program Loading in OSF/1", 1991.

David Aucsmith, et al., Proceedings of the 1996 Intel Software Developers' Conference, pp. 1–10, "Tamper Resistant Software: An Implementation", 1996.

* cited by examiner

FIG.1

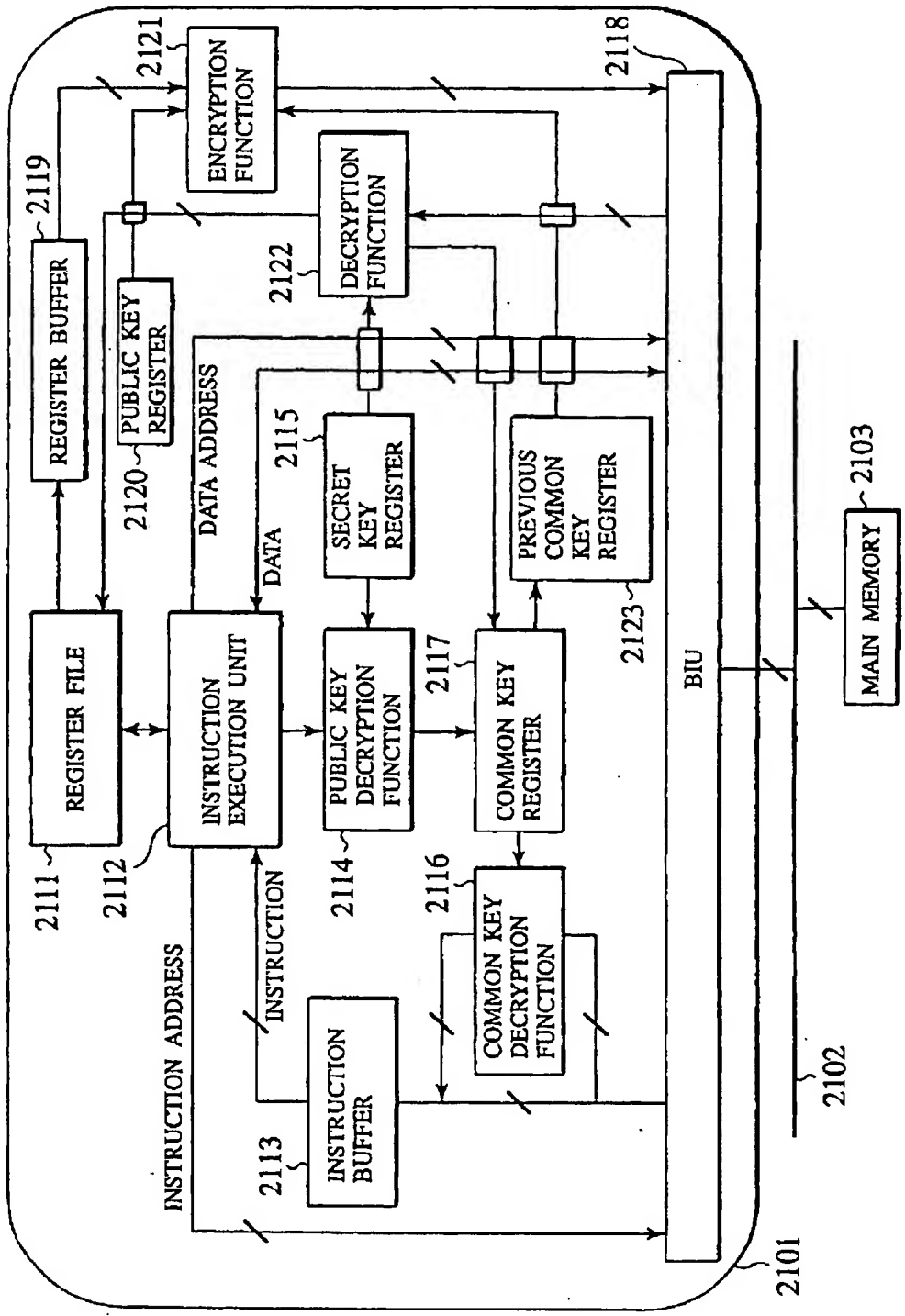

2101
2102
2103
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
REGISTER FILE
INSTRUCTION EXECUTION UNIT
INSTRUCTION BUFFER
PUBLIC KEY DECRYPTION FUNCTION
SECRET KEY REGISTER
COMMON KEY REGISTER
COMMON KEY DECRYPTION FUNCTION
PREVIOUS COMMON KEY REGISTER
BIU
MAIN MEMORY
REGISTER BUFFER
PUBLIC KEY REGISTER
ENCRYPTION FUNCTION
DECRYPTION FUNCTION
INSTRUCTION ADDRESS
INSTRUCTION
DATA ADDRESS
DATA

FIG.2

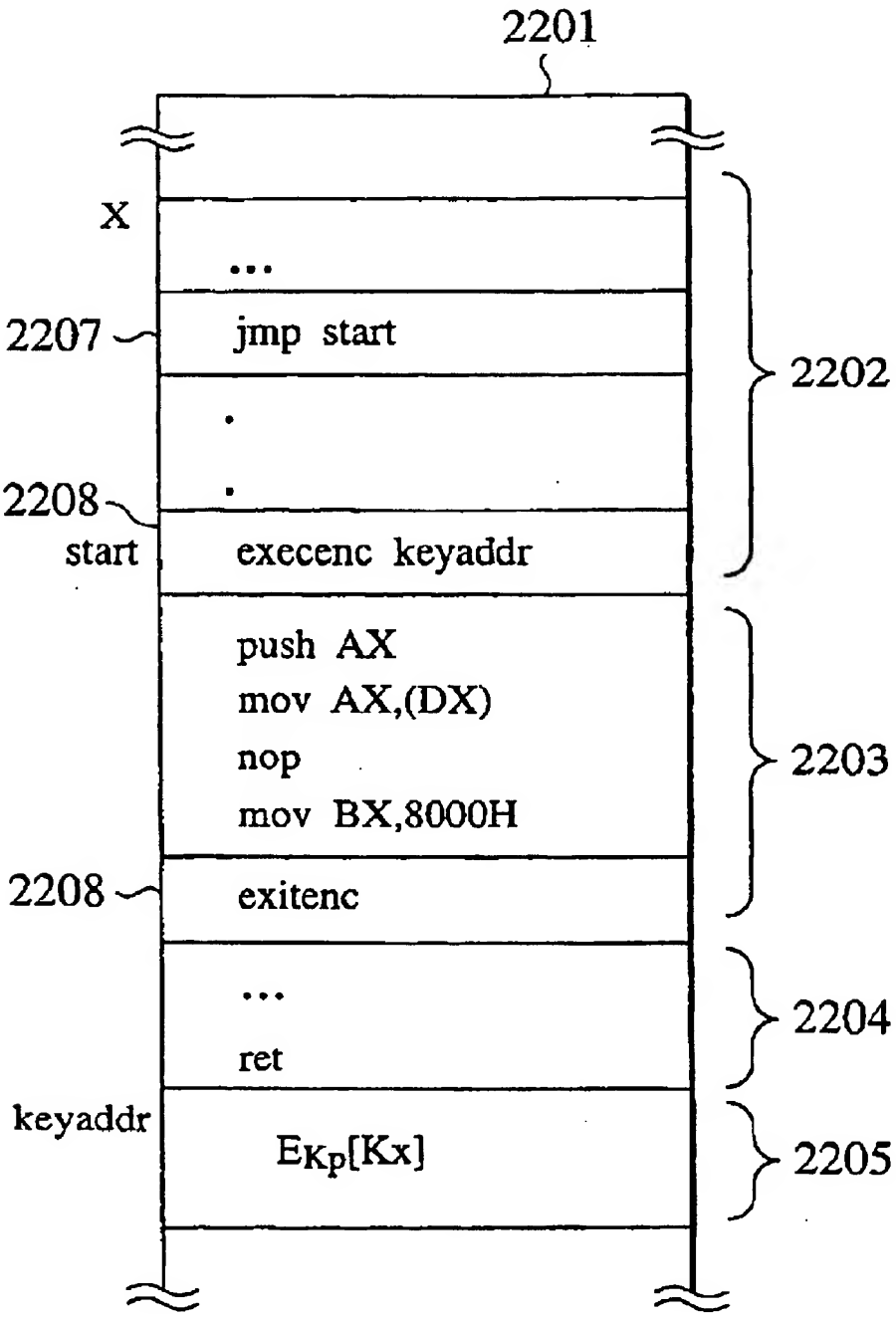
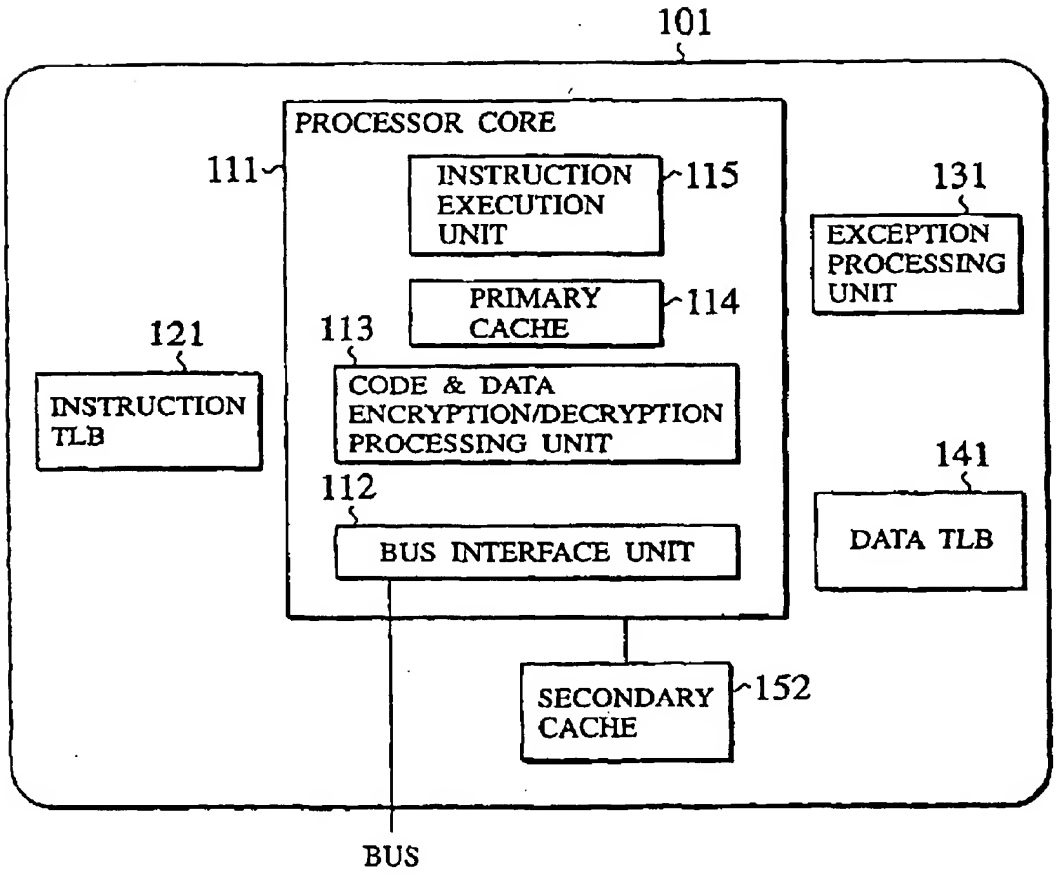

2201
X
...
2207
jmp start
2202
2208
start
execenc keyaddr
push AX
mov AX,(DX)
nop
mov BX,8000H
2203
2208
exitenc
...
ret
2204
keyaddr
EKp[Kx]
2205

FIG.3


101
PROCESSOR CORE
111
INSTRUCTION EXECUTION UNIT
115
131
EXCEPTION PROCESSING UNIT
PRIMARY CACHE
114
121
113
INSTRUCTION TLB
CODE & DATA ENCRYPTION/DECRYPTION PROCESSING UNIT
141
112
BUS INTERFACE UNIT
DATA TLB
SECONDARY CACHE
152
BUS

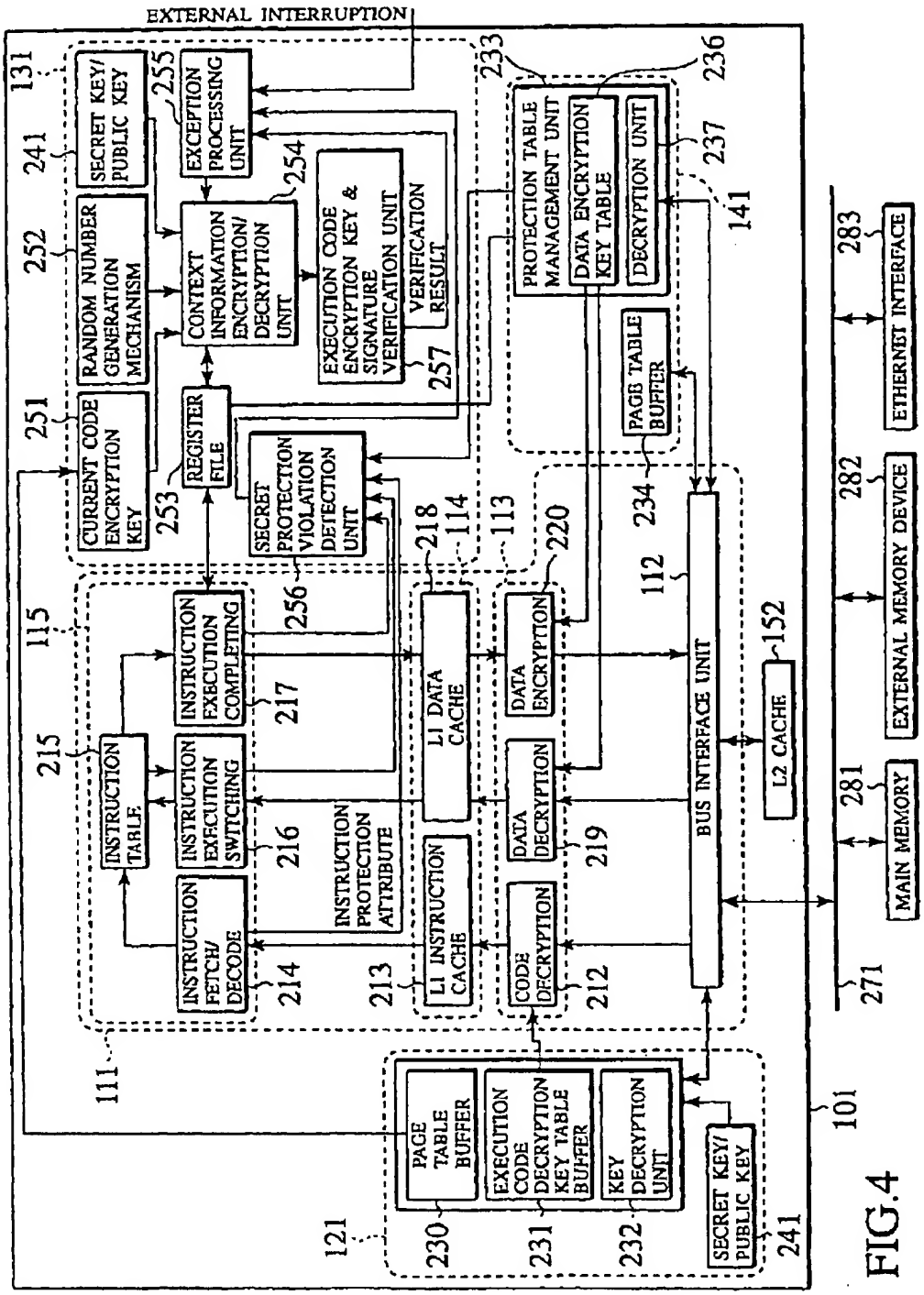

EXTERNAL INTERRUPTION
SECRET KEY/ PUBLIC KEY
EXCEPTION PROCESSING UNIT
RANDOM NUMBER GENERATION MECHANISM
CONTEXT INFORMATION ENCRYPTION/ DECRYPTION UNIT
EXECUTION CODE ENCRYPTION KEY & SIGNATURE VERIFICATION UNIT
VERIFICATION RESULT
PROTECTION TABLE MANAGEMENT UNIT
DATA ENCRYPTION KEY TABLE
DECRYPTION UNIT
CURRENT CODE ENCRYPTION KEY
REGISTER FILE
SECRET PROTECTION VIOLATION DETECTION UNIT
PAGE TABLE BUFFER
ETHERNET INTERFACE
INSTRUCTION TABLE
INSTRUCTION EXECUTION COMPLETING
INSTRUCTION EXECUTION SWITCHING
INSTRUCTION FETCH/ DECODE
INSTRUCTION PROTECTION ATTRIBUTE
L1 INSTRUCTION CACHE
L1 DATA CACHE
DATA DECRYPTION
DATA ENCRYPTION
CODE DECRYPTION
BUS INTERFACE UNIT
L2 CACHE
EXTERNAL MEMORY DEVICE
MAIN MEMORY
PAGE TABLE BUFFER
EXECUTION CODE DECRYPTION KEY TABLE BUFFER
KEY DECRYPTION UNIT
SECRET KEY/ PUBLIC KEY
111
113
114
115
121
131
141
152
212
213
214
215
217
218
219
220
230
231
232
233
234
236
237
241
251
252
253
254
255
256
257
271
281
282
283
101
112

FIG.4

FIG.5

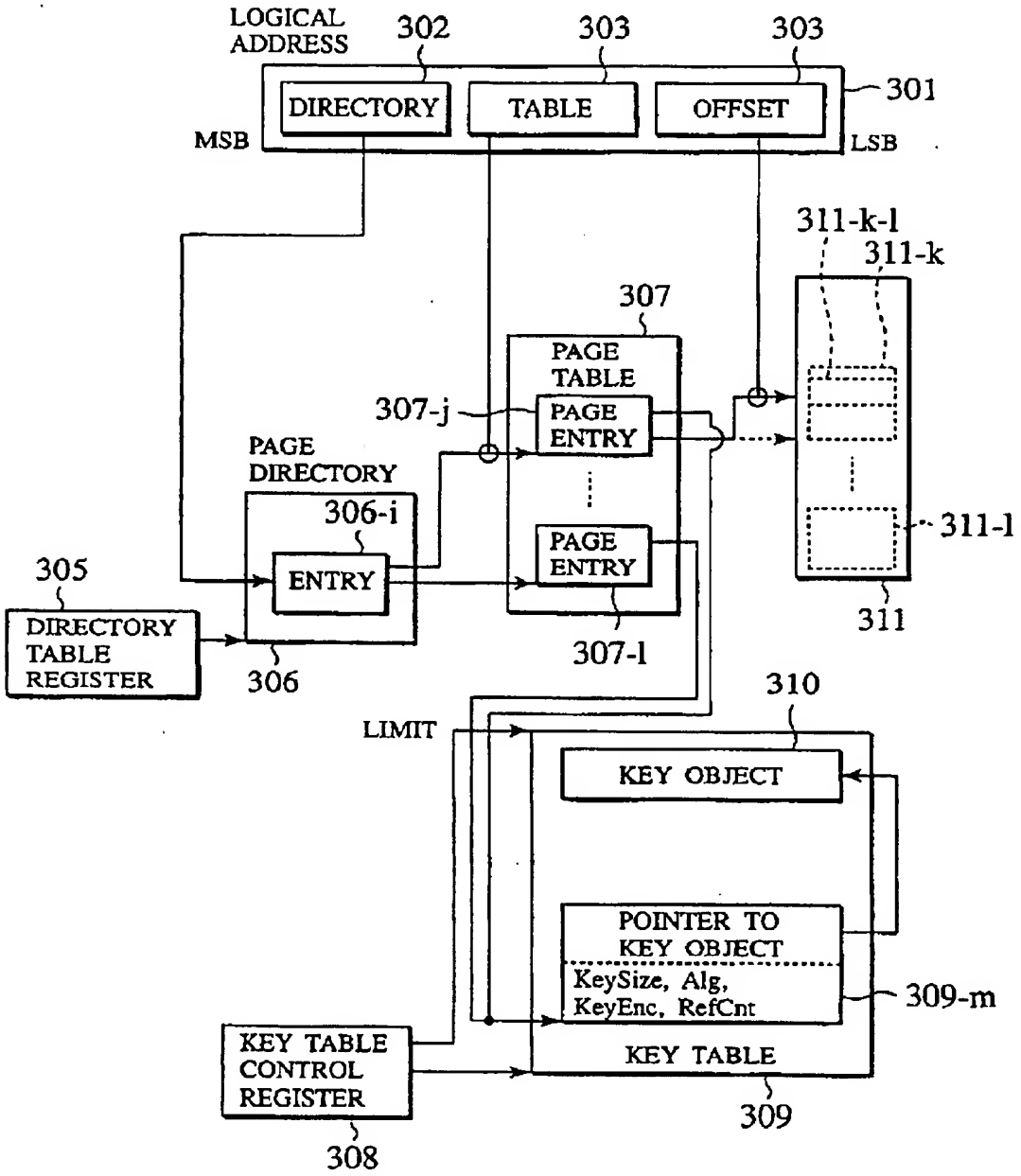

LOGICAL ADDRESS
302
303
303
301
DIRECTORY
TABLE
OFFSET
MSB
LSB
311-k-l
311-k
307
PAGE TABLE
307-j
PAGE ENTRY
PAGE DIRECTORY
306-i
ENTRY
PAGE ENTRY
311-l
305
DIRECTORY TABLE REGISTER
306
307-l
311
310
LIMIT
KEY OBJECT
POINTER TO KEY OBJECT
KeySize, Alg, KeyEnc, RefCnt
309-m
KEY TABLE CONTROL REGISTER
KEY TABLE
308
309

FIG.6

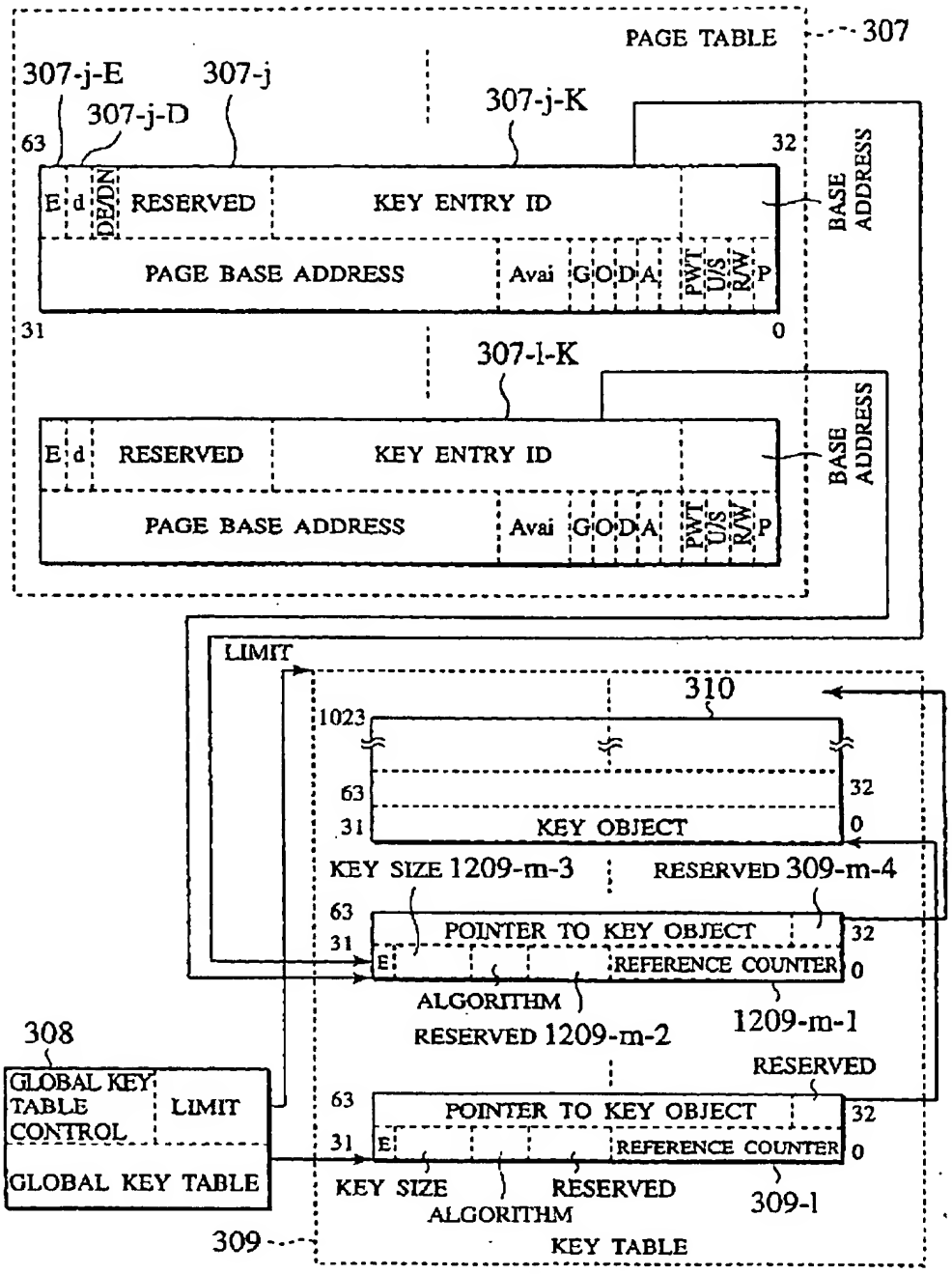

PAGE TABLE
307
307-j-E
307-j
307-j-D
307-j-K
63
32
E
d
DE/DN
RESERVED
KEY ENTRY ID
BASE ADDRESS
PAGE BASE ADDRESS
Avai
G
O
D
A
PWT
U/S
R/W
P
31
0
307-l-K
LIMIT
310
1023
63
32
31
0
KEY OBJECT
KEY SIZE 1209-m-3
RESERVED 309-m-4
POINTER TO KEY OBJECT
REFERENCE COUNTER
E
ALGORITHM
RESERVED 1209-m-2
1209-m-1
308
GLOBAL KEY TABLE CONTROL
LIMIT
GLOBAL KEY TABLE
RESERVED
KEY SIZE
ALGORITHM
309-1
309
KEY TABLE

FIG.7A

	0	1	2	3
0	A0	A1	A2	A3
1	B0	B1	B2	B3
2	C0	C1	C2	C3
3	D0	D1	D2	D3
4	E0	E1	E2	E3
5	F0	F1	F2	F3
6	G0	G1	G2	G3
7	H0	H1	H2	H3

BEFORE INTERLEAVING

FIG.7B

0	A0	B0	C0	D0
1	E0	F0	G0	H0
2	A1	B1	C1	D1
3	E1	F1	G1	H1
4	A2	B2	C2	D2
5	E2	F2	G2	H2
6	A3	B3	C3	D3
7	E3	F3	G3	H3

AFTER INTERLEAVING

FIG.8

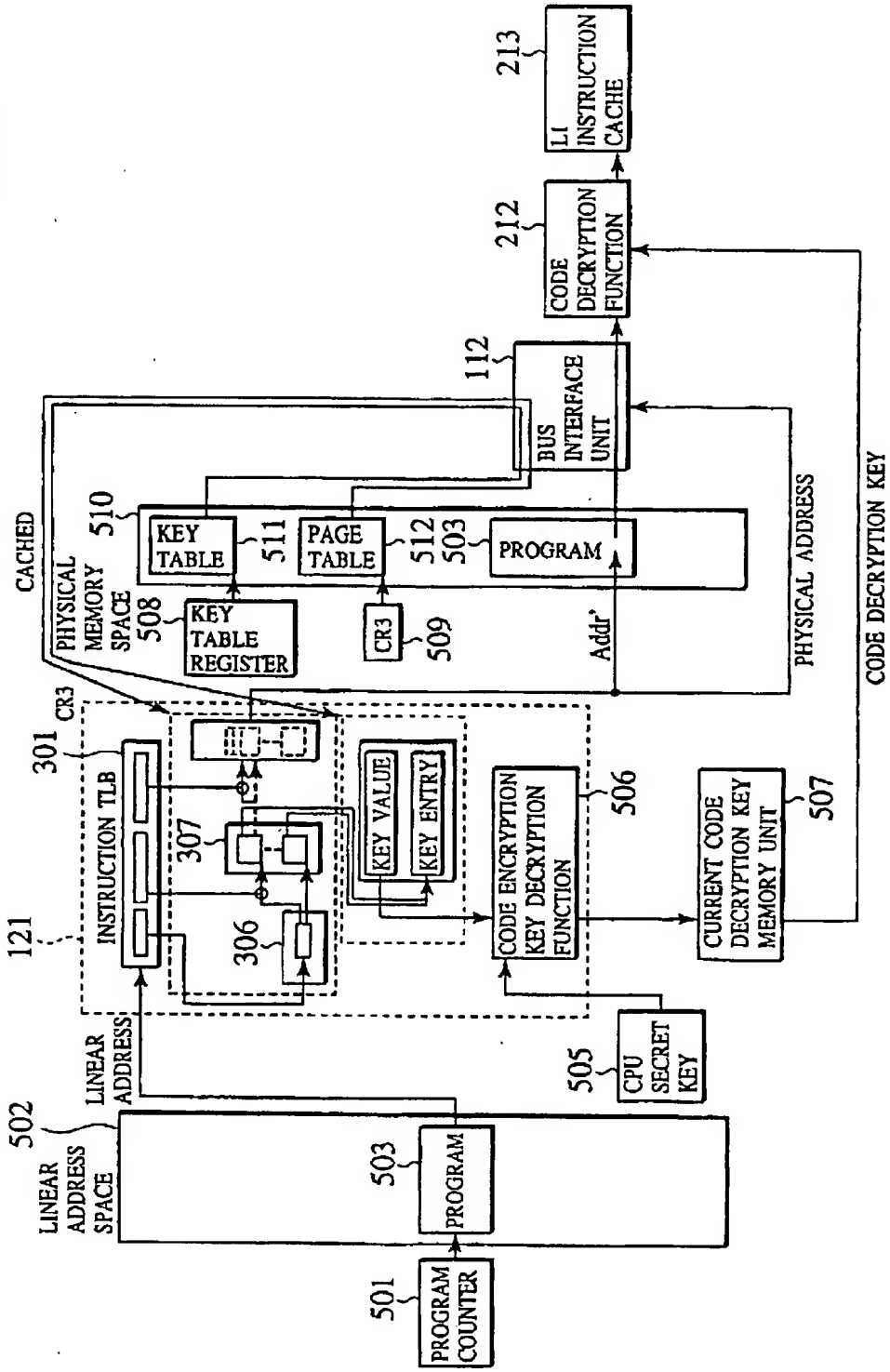

213
L1 INSTRUCTION CACHE
212
CODE DECRYPTION FUNCTION
112
BUS INTERFACE UNIT
CACHED
PHYSICAL MEMORY SPACE
510
KEY TABLE
511
PAGE TABLE
512
503
PROGRAM
508
KEY TABLE REGISTER
CR3
509
Addr'
PHYSICAL ADDRESS
CODE DECRYPTION KEY
301
CR3
INSTRUCTION TLB
307
KEY VALUE
KEY ENTRY
306
121
CODE ENCRYPTION KEY DECRYPTION FUNCTION
506
CURRENT CODE DECRYPTION KEY MEMORY UNIT
507
LINEAR ADDRESS
505
CPU SECRET KEY
502
LINEAR ADDRESS SPACE
503
PROGRAM
501
PROGRAM COUNTER

FIG.9

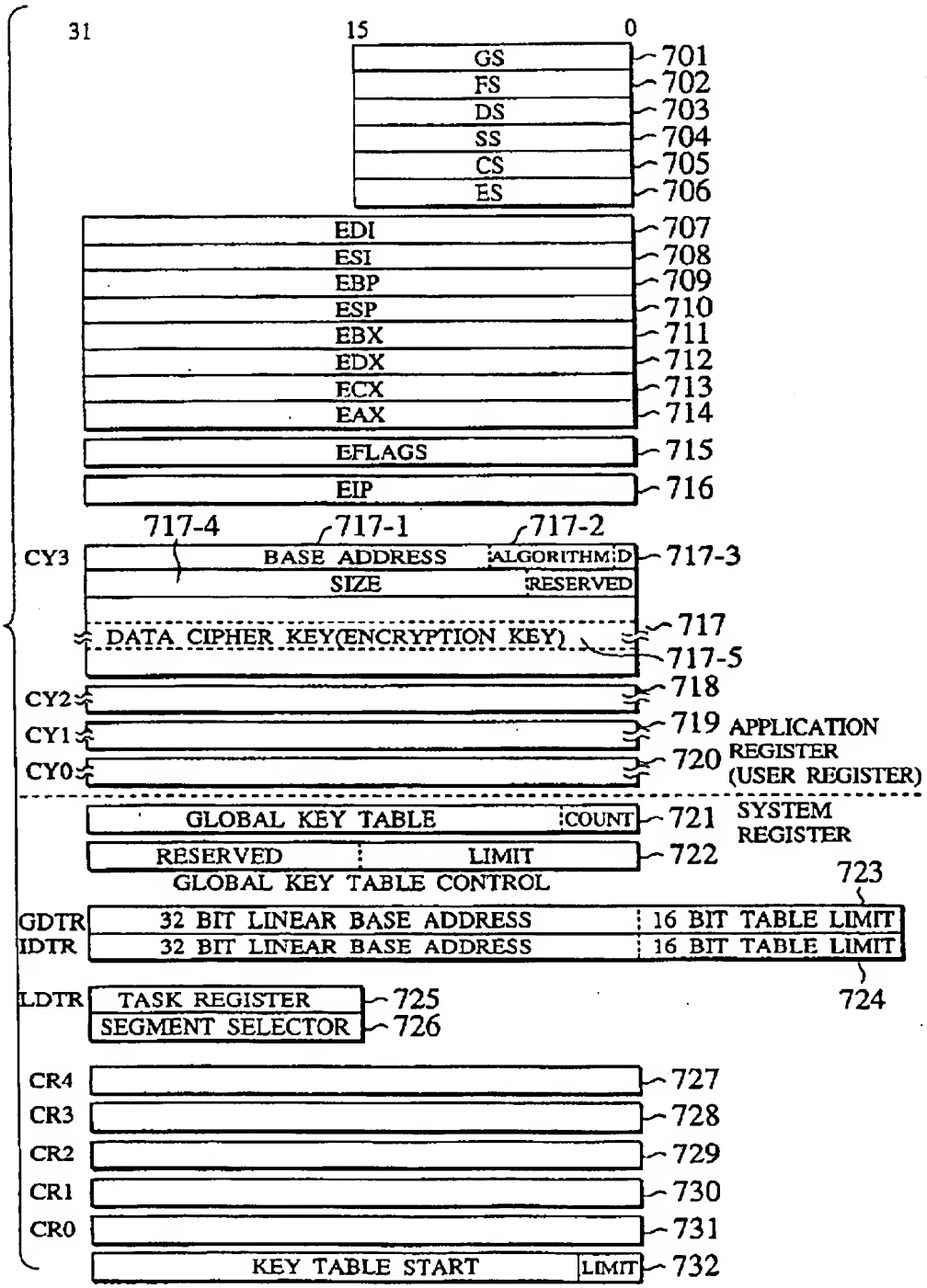

31
15
0
GS
FS
DS
SS
CS
ES
701
702
703
704
705
706
EDI
ESI
EBP
ESP
EBX
EDX
ECX
EAX
707
708
709
710
711
712
713
714
EFLAGS
715
EIP
716
717-4
717-1
717-2
CY3
BASE ADDRESS
ALGORITHM
D
717-3
SIZE
RESERVED
DATA CIPHER KEY(ENCRYPTION KEY)
717
717-5
CY2
718
CY1
719
APPLICATION REGISTER (USER REGISTER)
CY0
720
GLOBAL KEY TABLE
COUNT
721
SYSTEM REGISTER
RESERVED
LIMIT
722
GLOBAL KEY TABLE CONTROL
723
GDTR
32 BIT LINEAR BASE ADDRESS
16 BIT TABLE LIMIT
IDTR
32 BIT LINEAR BASE ADDRESS
16 BIT TABLE LIMIT
724
LDTR
TASK REGISTER
725
SEGMENT SELECTOR
726
CR4
727
CR3
728
CR2
729
CR1
730
CR0
731
KEY TABLE START
LIMIT
732

FIG.10

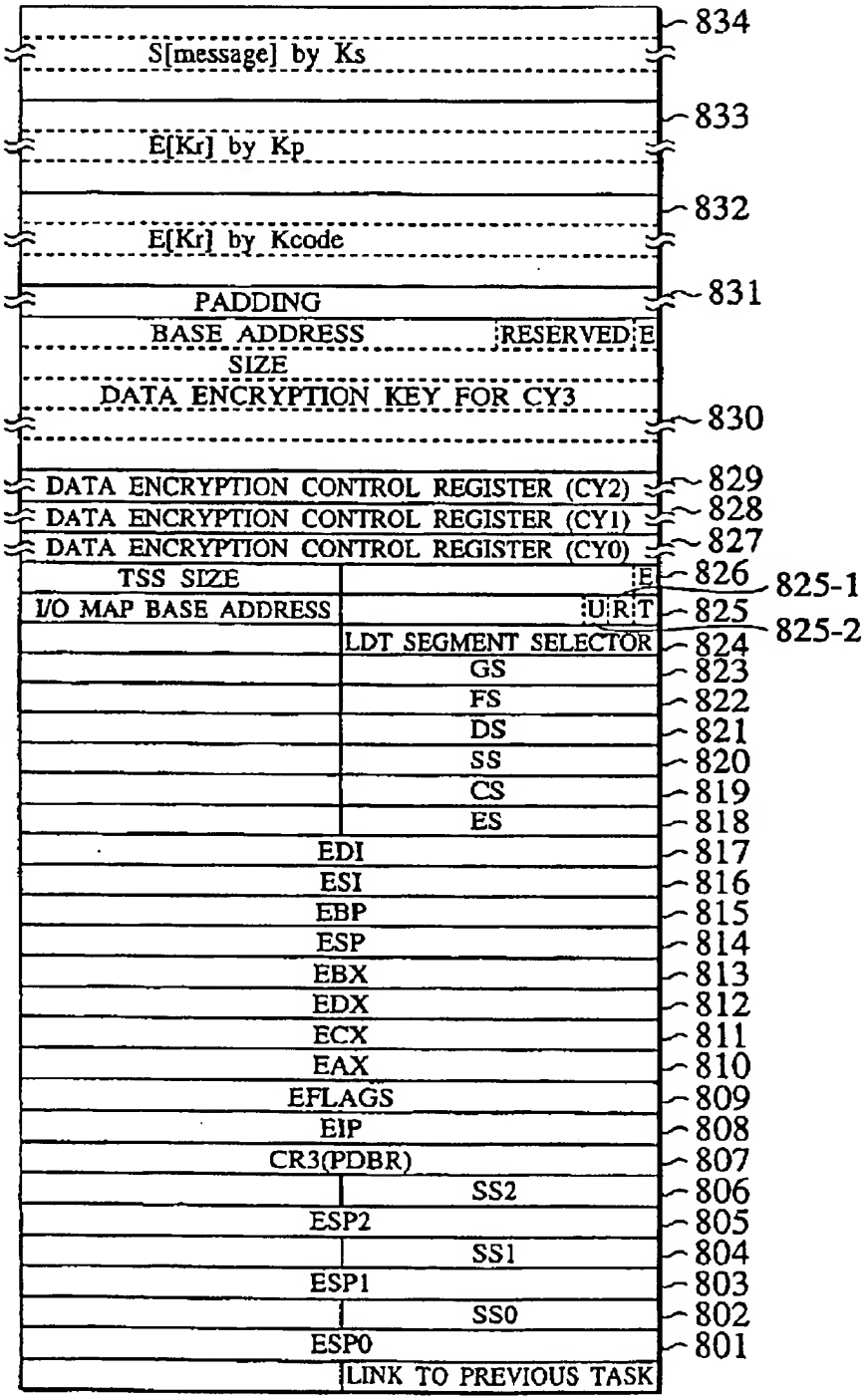
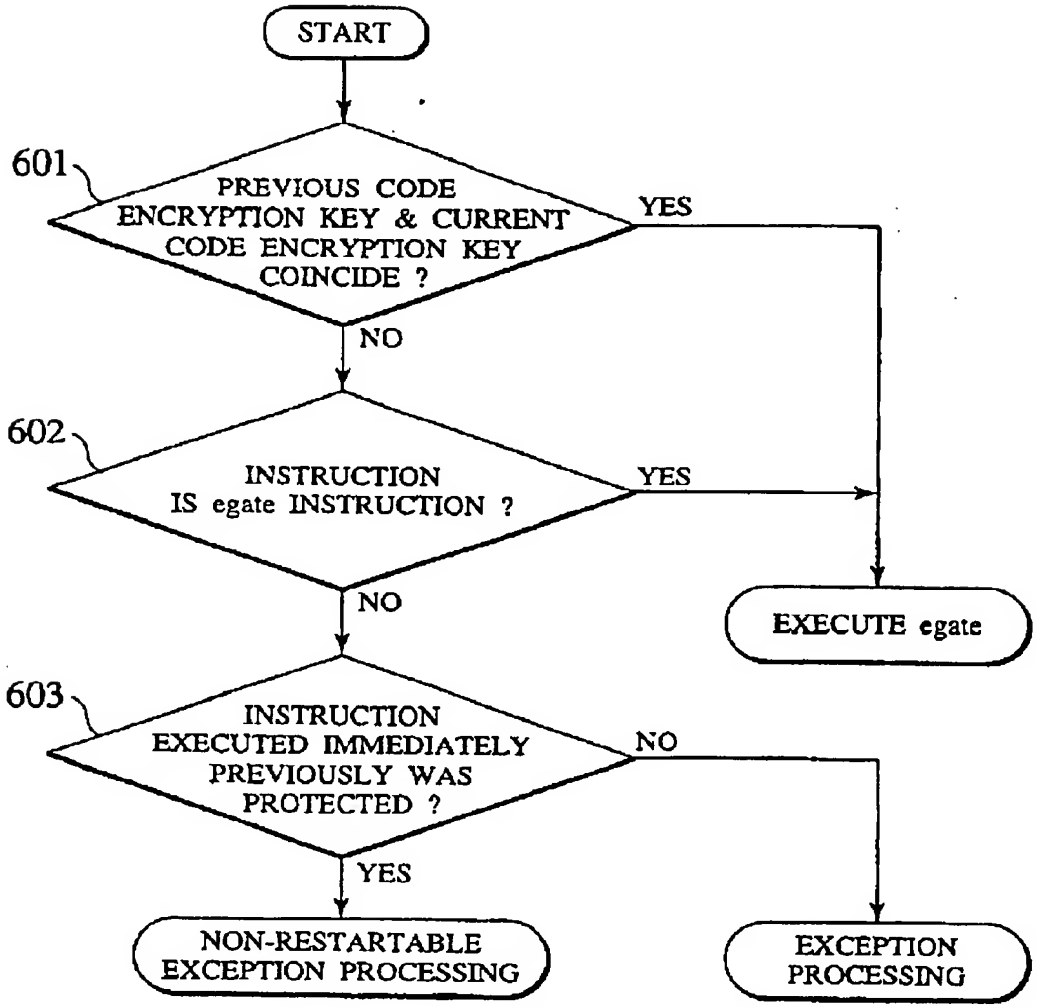

S[message] by Ks
834
E[Kr] by Kp
833
E[Kr] by Kcode
832
PADDING
831
BASE ADDRESS
RESERVED
E
SIZE
DATA ENCRYPTION KEY FOR CY3
830
DATA ENCRYPTION CONTROL REGISTER (CY2)
829
DATA ENCRYPTION CONTROL REGISTER (CY1)
828
DATA ENCRYPTION CONTROL REGISTER (CY0)
827
TSS SIZE
E
826
825-1
I/O MAP BASE ADDRESS
U
R
T
825
825-2
LDT SEGMENT SELECTOR
824
GS
823
FS
822
DS
821
SS
820
CS
819
ES
818
EDI
817
ESI
816
EBP
815
ESP
814
EBX
813
EDX
812
ECX
811
EAX
810
EFLAGS
809
EIP
808
CR3(PDBR)
807
SS2
806
ESP2
805
SS1
804
ESP1
803
SS0
802
ESP0
801
LINK TO PREVIOUS TASK

FIG.11


START
601
PREVIOUS CODE ENCRYPTION KEY & CURRENT CODE ENCRYPTION KEY COINCIDE ?
YES
NO
602
INSTRUCTION IS egate INSTRUCTION ?
YES
NO
EXECUTE egate
603
INSTRUCTION EXECUTED IMMEDIATELY PREVIOUSLY WAS PROTECTED ?
NO
YES
NON-RESTARTABLE EXCEPTION PROCESSING
EXCEPTION PROCESSING

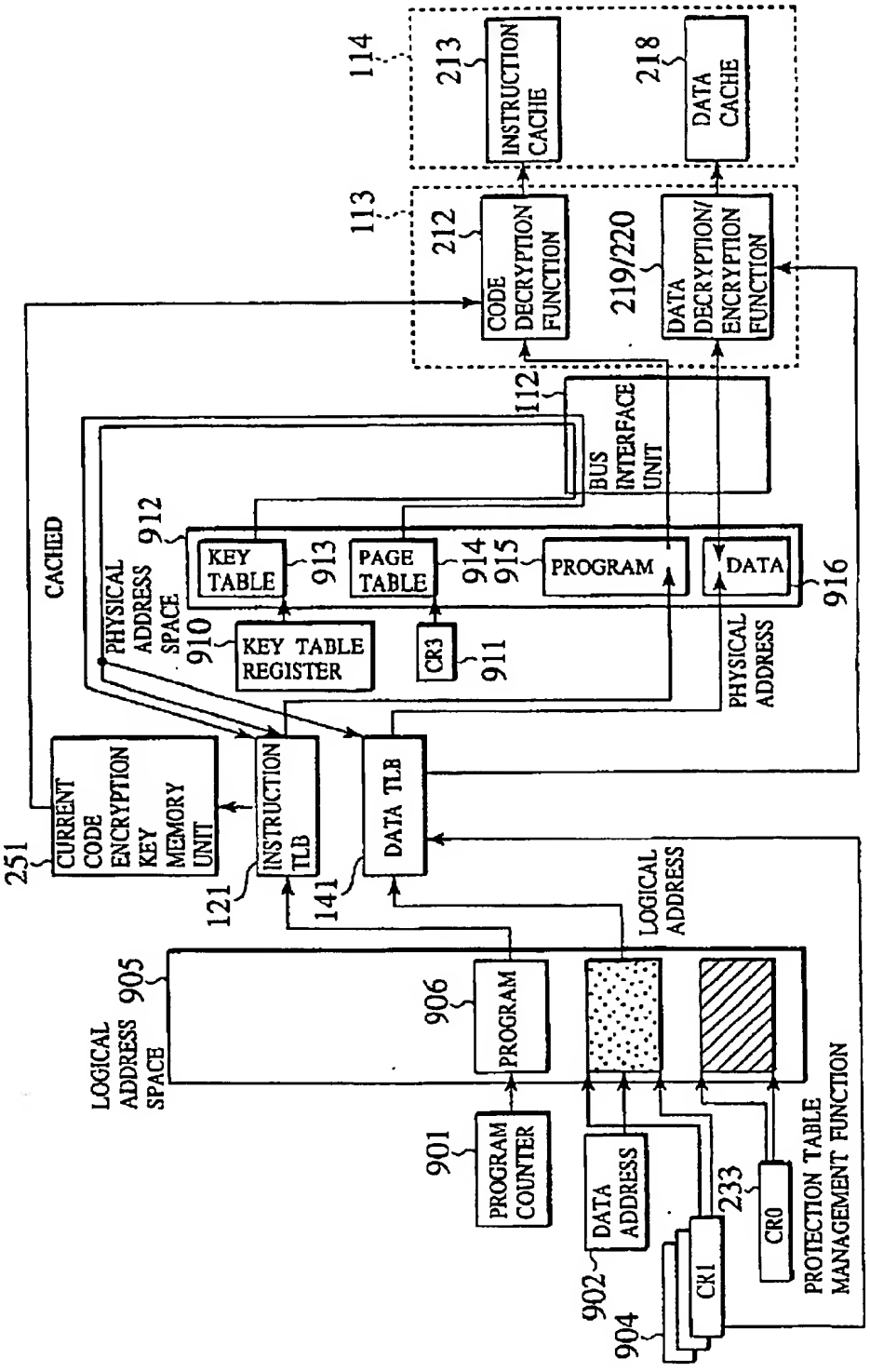
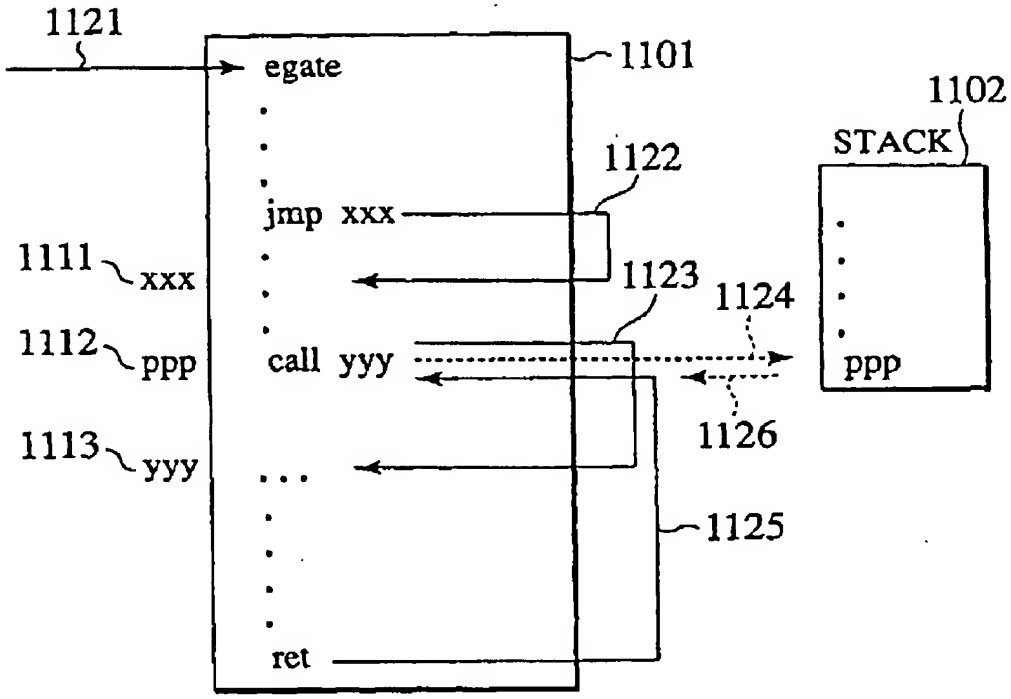

114
213
INSTRUCTION CACHE
218
DATA CACHE
113
212
CODE DECRYPTION FUNCTION
219/220
DATA DECRYPTION/ ENCRYPTION FUNCTION
112
BUS INTERFACE UNIT
CACHED
PHYSICAL ADDRESS SPACE
912
KEY TABLE
913
PAGE TABLE
914
915
PROGRAM
DATA
916
910
KEY TABLE REGISTER
CR3
911
PHYSICAL ADDRESS
251
CURRENT CODE ENCRYPTION KEY MEMORY UNIT
INSTRUCTION TLB
121
DATA TLB
141
LOGICAL ADDRESS
905
LOGICAL ADDRESS SPACE
906
PROGRAM
901
PROGRAM COUNTER
DATA ADDRESS
902
904
CR1
233
CR0
PROTECTION TABLE MANAGEMENT FUNCTION

FIG.12

FIG.13


1121
egate
1101
1102
STACK
1122
jmp xxx
1111
xxx
1123
1124
1112
ppp
call yyy
ppp
1126
1113
yyy
1125
ret

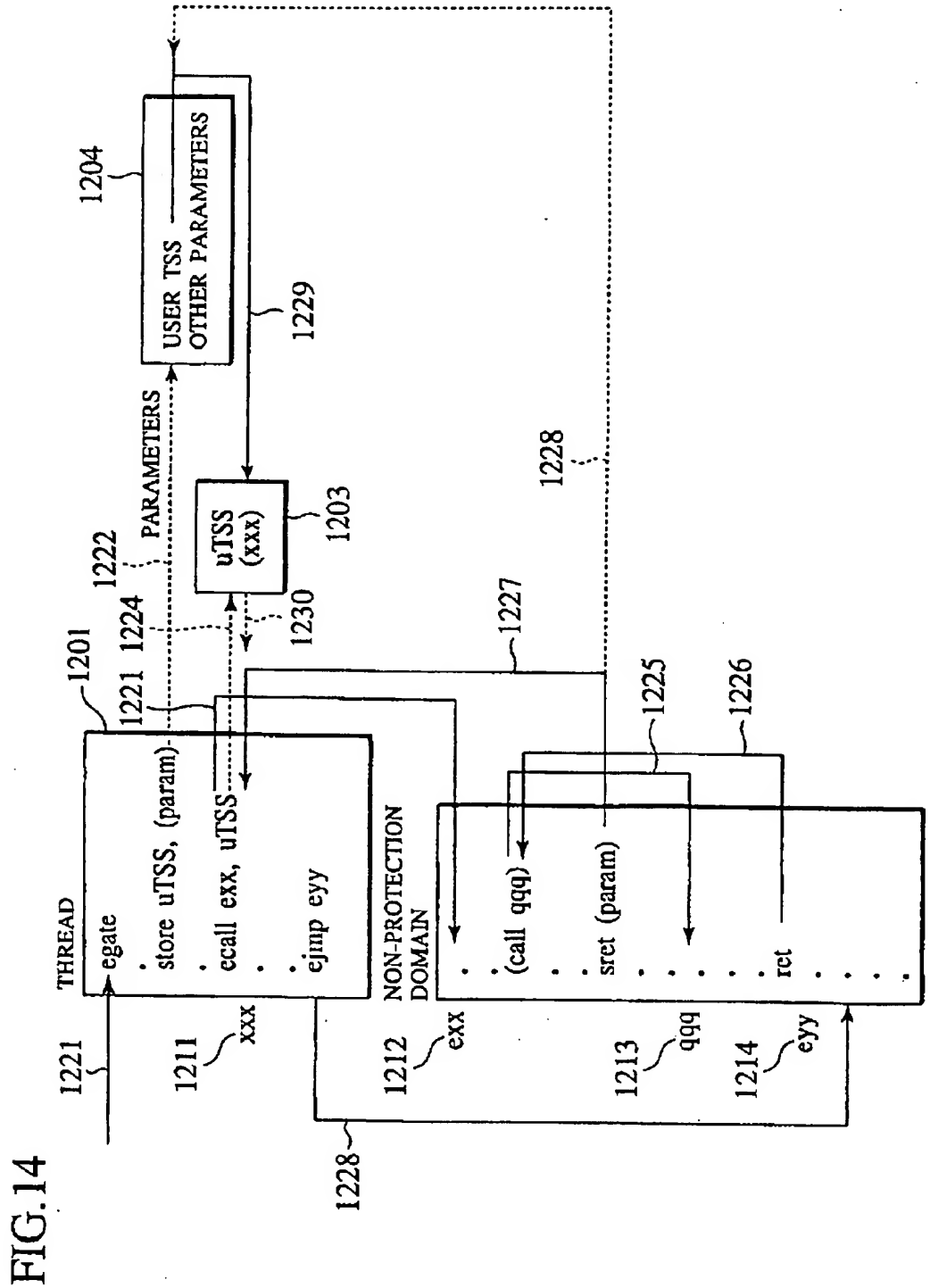

1204
USER TSS
OTHER PARAMETERS
1229
1228
PARAMETERS
1222
uTSS
(xxx)
1203
1224
1230
1227
1201
1221
1225
1226
THREAD
egate
store uTSS, (param)
ecall exx, uTSS
ejmp eyy
NON-PROTECTION
DOMAIN
(call qqq)
sret (param)
ret
1221
1211
xxx
1212
exx
1213
qqq
1214
eyy
1228

FIG. 14

FIG.15
PRIOR ART

31	15	0	
I/O MAP BASE ADDRESS		T	100
	LDR SEGMENT SELECTOR		96
	GS		92
	FS		88
	DS		84
	SS		80
	CS		76
	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3(PDBR)			28
	SS2		24
ESP2			20
	SS1		16
ESP1			12
	SS0		8
ESP0			4
	LINK TO PREVIOUS TASK		0

TAMPER RESISTANT MICROPROCESSOR

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a microprocessor that can prevent illegal alternation of execution codes and processing target data under a multi-task program execution environment.

2. Description of the Background Art

In recent years, the performance of a microprocessor has improved considerably such that the microprocessor is capable of realizing reproduction and editing of video images and audio sounds, in addition to the conventional functions such as computations and graphics. By implementing such a microprocessor in a system designed for end-user (which will be referred to as PC hereafter), the users can enjoy various video images and audio sounds on monitors. Also, by combing the function for reproducing video images and audio sounds with the computational power of the PC, the applicability to games or the like can be improved. Such a microprocessor is not designed for any specific hardware and can be implemented in a variety of hardwares so that there is an advantage that the users who already possess PCs can enjoy reproduction and editing of video images and audio sounds inexpensively by simply changing a microprocessor for executing programs.

In the case of handling video images and audio sounds on PCs, there arises a problem of a protection of the copyright of original images or music. In the MD or digital video playback devices, unlimited copies can be prevented by implementing a mechanism for preventing the illegal copying in these devices in advance. It is rather rare to attempt the illegal copying by disassembling and altering these devices, and even if such devices are made, there is a worldwide trend for prohibiting the manufacturing and sales of devices altered for the purpose of illegal copying by laws. Consequently, damages due to the hardware based illegal copying are not very serious.

However, image data and music data are actually processed on the PC by the software rather than the hardware, and the end-user can freely alter the software on the PC. Namely, if the user has some level of knowledge, it is quite feasible to carry out the illegal copying by analyzing programs and rewriting the executable software. In addition, there is a problem that the software for illegal copying so produced can be spread very quickly through media such as networks, unlike the hardware.

In order to resolve these problems, conventionally a PC software to be used for reproducing copyright protected contents such as commercial films or music has employed a technique for preventing analysis and alternation by encrypting the software. This technique is known as a tamper resistant software (see David Aucsmith et al., "Tamper Resistant Software: An Implementation", Proceedings of the 1996 Intel Software Developer's Conference).

The tamper resistant software technique is also effective in preventing illegal copying of valuable information including not only video and audio data but also text and know-how that is to be provided to a user through the PC, and protecting know-how contained in the PC software itself from analysis.

However, the tamper resistant software technique is a technique which makes analysis using tools such as deassembler or debugger difficult by encrypting a portion of the program that requires protection before the execution of the program starts, decrypting that portion immediately before executing that portion and encrypting that portion again immediately after the execution of that portion is completed. Consequently, as along as the program is executable by a processor, it is always possible to analyze the program by carrying out the analysis step by step starting from the start of the program.

This fact has been an obstacle for a copyright owner to provide copyright protected contents to a system for reproducing video and audio data using the PC.

The other tamper resistant software applications are also vulnerable in this regard, and this fact has been an obstacle to a sophisticated information server through the PC and an application of a program containing know-how of an enterprise or individual to the PC.

These are problems that equally apply to the software protection in general, but in addition, the PC is an open platform so that there is also a problem of an attack by altering the operating system (OS) which is intended to be a basis of the system's software configuration. Namely, a skilled and malicious user can alter the OS of his own PC to invalidate or analyze the copyright protection mechanisms incorporated in application programs by utilizing privileges given to the OS.

The current OS realizes the management of resources under the control of the computer and the arbitration of their uses by utilizing a privileged operation function with respect to a memory and an execution control function provided in CPU. Targets of the management include the conventional targets such as devices, CPU and memory resources, as well as QoS (Quality of Service) at network or application level. Nevertheless, the basics of the resource management are still allocations of resources necessary for the execution of a program. Namely, an allocation of a CPU time to the execution of that program and an allocation of a memory space necessary for the execution are the besics of the resource management. The control of the other devices, network and application QoS is realized by controlling the execution of a program that makes accesses to these resources (by allocating a CPU time and a memory space).

The OS has privileges for carrying out the CPU time allocation and the memory space allocation. Namely, the OS has a privilege for interrupting and restarting an application program at arbitrary timing and a privilege to move a content of a memory space allocated to an application program to a memory of a different hierarchical level at arbitrary timing, in order to carry out the CPU time allocation. The latter privilege is also used for the purpose of providing a flat memory space to the application by concealing (normally) hierarchical memory systems with different access speeds and capacities from the application.

Using these two privileges, the OS can interrupt an execution state of the application and take a snap shot of it at arbitrary timing, and restart it after making a copy of it or rewriting it. This function can also be used as a tool for analyzing secrets hidden in the application.

In order to prevent an analysis of the application on a computer, there are several known techniques for encrypting programs or data (Rampson, U.S. Pat. No. 4,847,902; Hartman, U.S. Pat. No. 5,224,166; Davis, U.S. Pat. No. 5,806,706; Takahashi et al., U.S. Pat. No. 5,825,878; Buer et al., U.S. Pat. No. 6,003,117; Japanese Patent Application Laid Open No. 11-282667 (1999), for example). However, these known techniques do not account for the protection of the program operation and the data secrecy from the above described privileged operations of the OS.

The conventional technique based on the x86 architecture of Intel Corporation (Hartman, U.S. Pat. No. 5,224,166) is a technique for storing the execution codes and data by encrypting them by using a prescribed encryption key Kx. The encryption key Kx is given in a form of $E_{Kp}[Kx]$ which is encrypted by using a public key Kp corresponding to a secret key Ks embedded in a processor. Consequently, only the processor that knows Ks can decrypt the encrypted execution codes on a memory. The encryption key Kx is stored in a register inside the processor called a segment register.

Using this mechanism, it is possible to protect the secrecy of the program codes from the user to some extent by encrypting the codes. Also, it becomes cryptographically difficult for a person who does not know the encryption key Kx of the codes to alter the codes according to his intention or newly produce codes that are executable when decrypted by using the encryption key Kx.

However, the system employing this technique has a drawback in that the analysis of the program becomes possible by utilizing a privilege of the OS called a context switching, without decrypting the encrypted execution codes.

More specifically, when the execution of the program is stopped by the interruption or when the program voluntarily calls up a software interruption command due to the system call up, the OS carries out the context switching for the purpose of the execution of the other program. The context switching is an operation to store an execution state (which will be referred to as a context information hereafter) of the program indicating a set of register values at that point into a memory, and restoring the context information of another program stored in the memory in advance into the registers.

FIG. 15 shows the conventional context storing format used in the x86 processor. All the contents of the registers used by the application are contained here. The context information of the interrupted program is restored into the registers when the program is restarted. The context switching is an indispensable function in order to operate a plurality of programs in parallel. In the conventional technique, the OS can read the register values at a time of the context switching, so that it is possible to guess most of the operations made by the programs if not all, according to how the execution state of that program has changed.

In addition, by controlling a timing at which the exception occurs by setting of a timer or the like, it is possible to carry out this processing at arbitrary execution point of the program. Apart from the interruption of the execution and the analysis, it is also possible to rewrite the register information by malicious intention. The rewriting of the registers can not only change the operation of the program but also make the program analysis easier. The OS can store arbitrary state of the application so that it is possible to analyze the operation of the program by rewriting the register values and operating the program repeatedly. In addition to the above described functions, the processor has a debugging support function such as a stepwise execution, and there has been a problem that the OS can analyze the application by utilizing all these functions.

As far as data are concerned, U.S. Pat. No. 5,224,166 asserts that the program can access the encrypted data only by the program execution using the encrypted code segment. Here, there is a problem that the encrypted data can be freely read by the encrypted program by using arbitrary key, regardless of the encryption key by which the program is encrypted, even when there are programs encrypted by using

mutually different encryption keys. This conventional technique does not account for the case where the OS and the application have their own secrets independently and the secret of the application is to be protected from the OS or a plurality of program providers have their own secrets separately.

Of course, it is possible to separate memory spaces among the applications and to prohibit accesses to a system memory by the applications by the protection function provided in the virtual memory mechanism even in the existing processor. However, as long as the virtual memory mechanism is under the management of the OS, the protection of the secret of the application cannot rely on the function under the management of the OS. This is because the OS can access data by ignoring the protection mechanism, and this privilege is indispensable in providing the virtual memory function as described above.

As another conventional technique, Japanese Patent Application Laid Open No. 11-282667 (1999) discloses a technique of a secret memory provided inside the CPU in order to store the secret information of the application. In this technique, a prescribed reference value is required in order to access data in the secret memory. However, this reference fails to disclose how to protect the reference value for obtaining the access right with respect to the secret data from a plurality of programs operating in the same CPU, especially the OS.

Also, in U.S. Pat. No. 5,123,045, Ostrovsky et al. disclose a system that presupposes the use of sub-processors having unique secret keys corresponding to the applications, in which the operation of the program cannot be guessed from the access pattern by which these sub-processors are accessing programs placed on a main memory. This is based on a mechanism for carrying out random memory accesses by converting the instruction system for carrying out operations with respect to the memory into another instruction system different from that.

However, this technique requires different sub-processors for different applications so that it requires a high cost, and the implementation and fast realization of the compiler and processor hardware for processing such instruction system are expected to be very difficult as they are quite different from those of the currently used processors. Besides that, in this type of processor, it becomes difficult to comprehend correspondences among the data contents and the operations even when the data and the operations of the actually operated codes are observed and traced so that the debugging of the program becomes very difficult, and therefore this technique has many practical problems, compared with the other conventional techniques described above in which the program codes and the data are simply encrypted, such as those of U.S. Pat. No. 5,224,166 and Japanese Patent Application Laid Open No. 11-282667.

SUMMARY OF THE INVENTION

Therefore the first object of the present invention is to provide a microprocessor capable of surely protecting both the internally executed algorithm and the data state inside a memory region from illegal analysis in the multi-task environment even when the execution is stopped by the interruption.

This first object is motivated by the fact that the conventional techniques are capable of protecting values of the program codes but are incapable of preventing the analysis utilizing the interruption of the program execution by the exception occurrence or the debugging function. Thus the

present invention aims at providing a microprocessor capable of surely protecting the codes even at a time of the program execution interruption, in which this protection is compatible with both the execution control function and the memory management function required by the current OS.

The second object of the present invention is to provide a microprocessor in which each program can secure a correctly readable/writable data region independently even when a plurality of programs encrypted by using different encryption keys are to be executed.

This second object is motivated by the fact that the conventional technique of U.S. Pat. No. 5,224,166 only provides a simple protection in which accesses to the encrypted data region by non-encrypted codes are prohibited, and it has been impossible for a plurality of programs to protect their own secrets independently. Thus the present invention also aims at providing a microprocessor which has a data region for protecting secret of each application from the OS when a plurality of applications have their respective (encrypted) secrets.

The third object of the present invention is to provide a microprocessor capable of protecting the protected attributes (i.e., encrypted attributed) of the above described data region from illegal rewriting by the OS.

This third object is motivated by the fact that the conventional technique of U.S. Pat. No. 5,224,166 has a drawback in that the OS can rewrite the encrypted attributes set in the segment register by interrupting the execution of the program using the context switching. Once the program is put in a state where data are written in a form of plaintext by rewriting the encrypted attributes, data will not written into a memory without encryption. Even if the application checks the segment register value at some timing, the result is the same if the register value is rewritten after that. Thus the present invention also aims at providing a microprocessor provided with a mechanism which is capable of prohibiting such an alteration or detecting such an alteration and taking appropriate measure against such an alteration.

The fourth object of the present invention is to provide a microprocessor capable of protecting the encrypted attributes from the so called chosen-plaintext attack of the cryptoanalysis theory, in which the program can use arbitrary value as the data encryption key.

The fifth object of the present invention is to provide a microprocessor provided with a mechanism for the program debugging and feedback. Namely, the present invention aims at providing a microprocessor in which the debugging of the program is carried out in plaintext and the feedback of information on defects is provided to a program code provider (program vendor) in the case of the execution failure.

The sixth object of the present invention is to provide a microprocessor capable of achieving the first to fifth objects described above in a form that realizes both a low cost and a high performance.

In order to achieve the first object, the first aspect of the present invention has the following features. The microprocessor which is formed as a single chip or a single package reads a plurality of programs encrypted by using code encryption keys that are different for different programs, from a memory (a main memory, for example) external of the microprocessor through a bus interface unit that provides a reading function. A decryption unit decrypts these plurality of read out programs by using respectively corresponding decryption keys, and an instruction execution unit executes these plurality of decrypted programs.

In the case of interrupting the execution of some program among the plurality of programs, a context information encryption/decryption unit that provides an execution state writing function encrypts information indicating a state of execution up to an interrupted point of the program to be interrupted and the code encryption key of this program, by using an encryption key unique to the microprocessor, and writes the encrypted information as a context information into a memory external of the microprocessor.

In the case of restarting the interrupted program, a verification unit that provides a restarting function decrypts the encrypted context information by using a unique decryption key corresponding to the unique encryption key of the microprocessor, and restarts the execution of the program only when the code encryption key contained in the decrypted context information (that is the code encryption key of the program scheduled to be restarted) coincides with the original code encryption key of the interrupted program.

In addition, in order to achieve the second and third objects, the microprocessor also has a memory region (a register, for example) inside the processor that cannot be read out to the external, and an encrypted attribute writing unit (an instruction TLB, for example) for writing encrypted attributes for the processing target data of the program into the internal memory. The encrypted attributes include the code encryption key of the program and an encryption target address range, for example). At least a part of these encrypted attributes is contained in the context information.

The context information encryption/decryption unit also attaches a signature based on a secret information unique to the microprocessor to the context information. In this case, the verification unit judges whether the signature contained in the decrypted context information coincides with the original signature based on the secret information unique to the microprocessor or not, and restarts the interrupted program only when they coincide.

In this way, the state of execution up to an interrupted point of the encrypted program is stored in the external memory as the context information, while the protected attributes of the execution processing target data are stored in the register inside the processor, so that the illegal alteration of the data can be prevented.

In order to achieve the fourth object, the second aspect of the present invention has the following features. The microprocessor that is formed as a single chip or a single package maintains a unique secret key therein that cannot be read out to the external. The bus interface unit that provides a reading function reads the code encryption key that is encrypted by using a unique public key of the microprocessor corresponding to the secret key in advance from a memory external of the microprocessor. A key decryption unit that provides a first decryption function decrypts the read out code encryption key by using the secret key of the microprocessor. The bus interface unit also reads out a plurality of programs encrypted by respectively different code encryption keys from an external memory. A code decryption unit that provides a second decryption function decrypts these plurality of read out programs. The instruction execution unit executes these plurality of decrypted programs.

In the case of interrupting the execution of some program among the plurality of programs, a random number generation mechanism generates a random number as a temporary key. The context information encryption/decryption unit writes a first value obtained by encrypting information indicating the execution state of the program to be interrupted by using the random number, a second value obtained

by encrypting this random number by using the code encryption key of the program to be interrupted, and a third value obtained by encrypting this random number by using the secret key of the microprocessor, into the external memory as the context information.

In the case of restarting the execution of the program, the context information encryption/decryption unit reads out the context information from the external memory, decrypts the random number of the third value contained in the context information by using the secret key, and decrypts the execution state information contained in the context information by using the decrypted random number. At the same time, the random number of the second value contained in the context information is decrypted by using the code encryption key of the program scheduled to be restarted. The random number obtained by decrypting the second value by using the code encryption key and the random number obtained by decrypting the third value by using the secret key are compared with the temporary key, and the execution of the program is restarted only when they coincide.

In this way, the context information indicating the state of execution up to an interrupted point is encrypted by using the random number that is generated at each occasion of the storing, and the signature using the secret key unique to the microprocessor is attached, so that the context information can be stored in the external memory safely.

In order to achieve the first to third and sixth objects, the third aspect of the present invention has the following features. The microprocessor that is formed as a single chip or a single package reads out a plurality of programs encrypted by using the encryption keys that are different for different programs, and executes them. This microprocessor has an internal memory (a register, for example) that cannot be read out to the external, and stores the encrypted attributes for data to be referred from each program (that is the processing target data) and the encrypted attribute specifying information into the register. The context information encryption/decryption unit writes a related information that is related to the encrypted attribute specifying information stored in the register and containing a signature unique to the microprocessor, into the external memory. A protection table management unit reads the related information from the external memory according to an address of the data to be referred by the program. The verification unit verifies the signature contained in the read out related information by using the secret key, and permits the data referring by the program according to the encrypted attribute specifying information and the read out related information only when that signature coincides with the signature unique to the microprocessor.

In this configuration, the information to be stored in the internal register is attached with the signature and stored into the external memory, and only the necessary portion is read out to the microprocessor. The signature is verified at a time of reading, so that the safety against the substitution can be secured. Even when the number of programs to be handled is increased and the number of the encrypted attributes is increased, there is no need to expand the memory region inside the microprocessor so that a cost can be reduced.

According to one aspect of the present invention there is provided a microprocessor having a unique secret key and a unique public key corresponding to the unique secret key that cannot be read out to external, comprising: a reading unit configured to read out a plurality of programs encrypted by using different execution code encryption keys from an external memory; a decryption unit configured to decrypt the plurality of programs read out by the reading unit by using respective decryption keys; an execution unit configured to execute the plurality of programs decrypted by the decryption unit; a context information saving unit configured to save a context information for one program whose execution is to be interrupted, into the external memory or a context information memory provided inside the microprocessor, the context information containing information indicating an execution state of the one program and the execution code encryption key of the one program; and a restart unit configured to restart an execution of the one program by reading out the context information from the external memory or the context information memory, and recovering the execution state of the one program from the context information.

Other features and advantages of the present invention will become apparent from the following description taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram showing a system incorporating a microprocessor according to the first embodiment of the present invention.

FIG. 2 is a diagram showing an entire memory space used in the microprocessor of FIG. 1.

FIG. 3 is a block diagram showing a basic configuration of a microprocessor according to the second embodiment of the present invention.

FIG. 4 is a block diagram showing a detailed configuration of the microprocessor of FIG. 3.

FIG. 5 is a diagram showing a page directory and a page table format used in the microprocessor of FIG. 3.

FIG. 6 is a page table and a key entry format used in the microprocessor of FIG. 3.

FIGS. 7A and 7B are diagrams respectively showing exemplary data before and after interleaving used in the microprocessor of FIG. 3.

FIG. 8 is a diagram showing a flow of information for a code decryption processing to be carried out in the microprocessor of FIG. 3.

FIG. 9 is a diagram showing a CPU register used in the microprocessor of FIG. 3.

FIG. 10 is a diagram showing a context saving format used in the microprocessor of FIG. 3.

FIG. 11 is a flow chart for a protection domain switching procedure to be carried out in the microprocessor of FIG. 3.

FIG. 12 is a diagram showing a flow of information for data encryption and decryption processing to be carried out in the microprocessor of FIG. 3.

FIG. 13 is a diagram conceptually showing a process of execution control within a protection domain by the microprocessor of FIG. 3.

FIG. 14 is a diagram conceptually showing a process of call up and branching from a protection domain to a non-protection domain by the microprocessor of FIG. 3.

FIG. 15 is a diagram showing a context saving format used in a conventional processor.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Referring now to FIG. 1 and FIG. 2, the first embodiment of a tamper resistant microprocessor according to the present invention will be described in detail.

This first embodiment is directed to a microprocessor for protecting secrets of the program instructions (execution codes) and the context information (execution state) which are to be provided in encrypted forms by using the public key (asymmetric key) cryptosystem, from a user of a target system.

FIG. 1 shows the target system, where a microprocessor **2101** of the target system is connected to a main memory **2103** through a bus **2102**.

As shown in FIG. 1, in this embodiment, the microprocessor **2101** has a register file **2111**, an instruction execution unit **2112**, an instruction buffer **2113**, a public key decryption function **2114**, a secret key register **2115**, a common key decryption function **2116**, a common key register **2117**, a BIU (Bus Interface Unit) **2118**, a register buffer **2119**, a public key register **2120**, an encryption function **2121**, a decryption function **2122**, and a previous common key register **2123**, which will be described in further detail below.

First, the terms to be used in the following description will be described, and the operation of general operating system (OS) and application programs will be described briefly. A program is a set of data and a series of machine language instructions written for some specific purpose. The OS is a program for managing resources of the system, and the application is a program to be operated under the resource management of the OS. This embodiment presupposes the multi-task system, so that a plurality of application programs will be operated in a quasi parallel manner under the management of the OS. Each one of these programs that are operated in the quasi parallel manner will be referred to as a process. There are cases where a set of processes for executing the processes for the same purpose will be referred to as a task.

The instructions and data of the application program are usually stored in files on a secondary memory. They are arranged on a memory by a loader of the OS and executed as a process. The execution of the program is often interrupted by an exception (or interruption) processing of the processor caused by input/output or the like. A program for carrying out the exception processing will be referred to as an exception handler. The exception handler is usually set up by the OS. The OS can process an exception request from the hardware, interrupt the operation of the application and restart or start the operation of another application at arbitrary timing. The interruptions of the process include transitory cases where the execution of the original process is restarted without switching processes after the execution of the exception handler, and cases requiring the process switching. Examples of the former include a simple timer increment and examples of the latter include a virtual memory processing due to the page exception.

The object of this embodiment is to protect the program instructions (execution codes) and the execution state from a user of the target system who can freely read the main memory of the target system and freely alter the OS program or application programs.

The basic features for achieving this object are the access control with respect to the information storage inside the processor and the encryption based on the information listed below.

(1) A common key Kx selected by a program creator. The application program will be encrypted by the secret key cryptosystem using this key.

(2) A pair of a unique public key Kp and a unique secret key Ks provided inside the processor. The public key can be read out by the program by using instructions.

(3) An encryption key information in which the common key Kx of the program is encrypted by using the public key Kp of the processor.

[Execution of a Plaintext Program]

This processor is capable of executing a program with coexisting plaintext instructions and encrypted instructions which is placed on the main memory. Here the operation inside the CPU for the execution of a plaintext program will be described with references to FIG. 1 and a memory arrangement shown in FIG. 2.

FIG. 2 shows an entire memory space **2201**, in which programs are placed in regions **2202** to **2204** on the main memory, where regions **2202** and **2204** are plaintext regions while a region **2203** is an encrypted region. A region **2205** stores a key information to be used in decrypting the region **2203**.

The execution of the program is started as the control is shifted from the OS by an instruction for jump to a top X of the program or the like. The instruction execution unit **2112** executes the instruction for jump to X, and outputs an address of the instruction to the BIU **2118**. The content of the address X is read through the bus **2102**, sent from the BIU **2118** to the instruction buffer **2113**, and sent to the instruction execution unit **2112** where the instruction is executed. Its operation result is reflected in the register file **2111**. When the operation target is reading/writing with respect to an address on the main memory **2103**, its address value is sent to the BIU **2118**, that address is outputted from the BIU **2118** to the bus **2102**, and data reading/writing with respect to the memory is carried out.

The instruction buffer **2113** has a capacity for storing two or more instructions, and the instructions corresponding to a size of the instruction buffer **2113** are collectively read out from the main memory **2103**.

[Execution of Encrypted Instructions]

Next, the case of executing an encrypted instruction will be described. The processor of this embodiment has two states including the execution of plaintext instructions and the execution of encrypted instructions, and two types of instructions for controlling these states are provided. One is an encryption execution start instruction for making a transition from the execution of plaintext instructions to the execution of encrypted instructions, and another is a plaintext return instruction for making a reverse transition.

[Encryption Execution Start Instruction]

The encryption execution start instruction is denoted by the following mnemonic "execenc" and takes one operand:

```
execenc keyaddr
```

where "keyaddr" indicates an address where the key information to be used in decrypting the subsequent instructions is stored.

[Key Information]

Here, the key information and the program encryption will be described. The encrypted region **2203** comprises a sequence of encrypted instructions. The instructions are subdivided into blocks in units of a prefetch queue size and encrypted by the secret key algorithm such as DES (Data Encryption Standard) algorithm. A key to be used in this encryption will be denoted as Kx hereafter. Since the secret key algorithm is used, the same key Kx is also used for the decryption.

If this Kx is placed on the main memory in a plaintext form, a user who can operates the OS of the target system can easily read it and analyze the encrypted program. In

order to prevent this, $E_{Kp}[Kx]$ obtained by encrypting Kx by using the public key Kp of the processor will be placed in the region **2205** of the memory. A top address of this region is indicated by "keyaddr".

It is cryptographically (computationally) impossible to decrypt Kx from $E_{Kp}[Kx]$ unless one knows Ks corresponding to the public key Kp. Consequently, the secret of the program will never be leaked to the user as long as the user of the target system does not know Ks. This Ks is stored in a form that cannot be read out from the external, inside the processor. The processor can decrypt Kx internally without allowing the user to learn about it, and the processor can also decrypt the encrypted program by using Kx and execute it.

In the following, the encryption execution start instruction and the subsequent the execution of the encrypted instruction will be described in detail. By the execution of the Jump instruction in a region **2207**, the control is shifted to the encryption execution start instruction at the address "start". At the address indicated by the operand "keyaddr" of the encryption execution start instruction, the content of the specified region **2205** is read out to the instruction execution unit **2112** of the processor as data. The instruction execution unit **2112** sends this data $E_{Kp}[Kx]$ to the public key decryption function **2114**. The public key decryption function **2114** takes out Kx by decrypting $E_{Kp}[Kx]$ by using a secret key Ks unique to the processor which is stored in the secret key register **2115**, and stores it in the common key register **2117**. Then, the processor enters the encrypted instruction execution state.

Here, it is assumed that the processor package is manufactured such that the contents stored in the secret key register **2115** and the common key register **2117** cannot be read out to the external by the program or the debugger of the processor chip.

By executing the encryption execution start instruction, the key to be used in decrypting the subsequent instructions is stored into the common key register **2117**, and the processor is entered into the encrypted instruction execution state. When the processor is in the encrypted instruction execution state, the instructions read from the main memory **2103** are sent from the BIU **2118** to a common key decryption function **2116**, decrypted by using the key information stored in the common key register **2117** and stored into the instruction buffer **2113**.

In this embodiment, the program encrypted by using the key Kx which is stored in the region **2204** next to the encryption execution start instruction will be decrypted, stored in the instruction buffer **2113**, and executed. The reading is carried out in units of a size of the instruction buffer **2113**. FIG. **2** shows an exemplary case where the size of the instruction buffer **2113** is 64 bits, and four instructions of 16 bits size each are collectively read out to the instruction buffer **2113**.

[Plaintext Return Instruction]

The processor in the encrypted instruction execution state returns to the plaintext instruction execution state by the execution of the plaintext return instruction.

The plaintext return instruction is denoted by the following mnemonic:

```
exitenc
```

which takes no operand. By execution of this instruction, the reading of the instructions from the main memory **2103** is carried out through a path that does not pass through the common key decryption function **2116**, and the processor returns to the execution of the plaintext instructions.

Note that when the encryption execution start instruction is executed again during the execution of the encrypted instruction, the instruction decryption key is changed such that the subsequent instructions are decrypted by using a different key and executed.

[Context Saving and Attack Against It]

Next, the safe saving of the execution state in order to protect the secret of the application program in the multi-task environment will be described.

The register file **2111** of this processor has 32 general purpose registers (R0 to R31). R31 is used as a program counter. The contents of the general purpose registers are stored in the register file **2111**. When the exception occurs during the execution of the encrypted program as described above, the contents of the register file **2111** are moved to the register buffer **2119**, and the contents of the register file **2111** are initialized by a prescribed value or a random number. Then, the value of the common key used for decryption of the encrypted program is stored in the previous common key register **2123**. Only when these two types of initialization are completed, the control is shifted to the exception handler and the instructions of the exception handler are executed. The instructions of the exception handler are assumed to be non-encrypted.

By this register file initialization function, in the processor of this embodiment, the reading of the register values processed by the encrypted program by the exception handler program is prevented even in the case where the control is shifted to the exception handler as an exception occurs during the execution of the encrypted program. At the same time, the contents of the register file **2111** are saved in the register buffer **2119**, and there is a function for recovering the register buffer contents and for storing them into the memory as will be described below, so as to enable the restart of the encrypted program.

Now, the register contents stored in the register buffer **2119** cannot be read out directly from the non-encrypted program of the exception handler. The non-encrypted program of the exception handler is only allowed to perform the following two operations with respect to the register buffer **2119**.

(1) Recover the register buffer contents and restart the execution of the original encrypted program.

(2) Encrypting the register buffer contents and store them into the memory, and execute the OS program or another encrypted program.

In the case of (1), when the exception handler processing such as the increment of the counter is finished, the exception handler issued a "cont" (continue) instruction. When the "cont" instruction is executed, the contents of the register buffer **2119** and the previous common key register **2123** are recovered in the register file **2111** and the common key register **2117**, respectively. The program counter is contained in the register file **2111**, so that the execution of the encrypted program is restarted by setting the control back to a point where the execution of the encrypted program was interrupted. For the decryption of the encrypted program after the restart, the value recovered from the previous common key register **2123** will be used. Similarly as the contents of the register buffer **2119**, the program cannot rewrite the previous common key register **2123** explicitly.

The case of (2) corresponds to the case where the process switching occurs at a timing of the execution of the exception handler. In this case, the exception handler or a task dispatcher of the processor issues a "savereg" (save register) instruction for saving the contents of the register buffer **2119** into the memory. This "savereg" instruction is denoted by the following mnemonic:

```
savereg dest
```

and takes one operand "dest" indicating an address to which the register buffer contents are to be saved.

When the "savereg" instruction is issued, the contents of the register buffer **2119** and the previous common key register **2123** are encrypted by the encryption function **2121** by using the public key Kp of the processor stored in the public key register **2120**, and saves at an address on the main memory **2103** specified by "dest" through the BIU **2118**. The main memory **2103** is outside the processor so that it has a possibility of being accessed by the user, but these contents are encrypted by the public key of the processor so that the user who does not know the secret key of the processor cannot learn the register buffer contents.

After the register buffer contents are saved, the OS activates another encrypted program by the method described above. If another encrypted program is activated without saving the register buffer contents, the register buffer contents would be rewritten to those of another encrypted program when the execution of another encrypted program is interrupted, and it would become impossible to restart the original encrypted program as the register buffer contents for the original encrypted program are lost.

Here, the number of the register buffer is assumed to be one, but it is also possible to provide a plurality of register buffers so as to be able to deal with multiple exceptions.

[Recovery Procedure]

Next a procedure for recovering the saved execution state will be described.

At a time of restarting the interrupted application, a dispatcher of the OS issues a "revrreg" (recover register) instruction. This "revrreg" instruction is denoted by the following mnemonic:

```
revrreg addr
```

and takes one operand "addr" indicating an address at which the execution state is saved.

When the "revrreg" instruction is issued, the encrypted execution state information is taken out from the address of the memory specified by "addr" by the BIU **2118** of the processor, decrypted by using the secret key Ks of the processor by the decryption function **2122**, and the register information is recovered in the register file **2111** while the program decryption key is recovered in the common key register **2117**. When the recovery is completed, the execution of the interrupted program is restarted from a point indicated by the program counter. At this point, the key Kx recovered from the execution state information will be used for decryption of the encrypted program.

The detail of the saving and the recovery of the execution state in relation to the interruption of the encrypted program due to exception has been described above. As already described above, the encrypted programs are safe against attacks from the user who can operate the OS of the target system.

Next, the safety of the above described scheme against two types of attacks against the execution state will be described.

[Attacks Against the Execution State]

There are two types of attacks against the execution state that is generated in a process of the application execution. One is the peeping of the saved execution state by an attacker, and the other is the rewriting of the execution state to a desired value by an attacker.

Here, the following two terms for expressing the illegal accesses to the execution state will be defined. First, the program that has generated the execution state will be referred to as an original program for that execution state. The original program can be restarted by recovering the execution state in the registers. On the other hand, programs other than the program that has generated the execution state, that is programs encrypted by encryption keys different from that of the original program or plaintext programs, will be referred to as other programs.

The illegal accesses or the attacks with respect to the execution state generated by some original program are defined as an act of directly analyzing the execution state on the memory by some method independently from the operation of the processor by a third party who does not know the encryption key of the original program, or an act of analyzing the execution state or rewriting the execution state to a desired value by a third party utilizing the other programs operated on the same processor.

In the microprocessor of this embodiment, the execution state is protected by the following three types of mechanisms so as to prevent the illegal accesses utilizing the access to the memory external of the processor or the other programs.

First, in this embodiment, the register information is saved in the register buffer **2119** when the execution of the encrypted program is interrupted. Then, the register buffer **2119** and the previous common key register **2123** cannot be accessed by any methods other than that using the "revrreg" instruction or the "savereg" instruction, so that the other programs cannot read their contents freely.

In the conventional processor, the register contents at a time of the exception occurrence can be freely read by the exception handler program. In the microprocessor of this embodiment, the register contents are saved in the register buffer **2119** so as to prohibit the reading from the other programs, and the instruction for saving the register buffer contents by encrypting them by using the public key of the processor is provided so as to prevent the peeping of the execution state saved on the memory by the user of the system.

The second attacking method includes a method for reading values of the registers contained in the execution state by placing the instruction of some other program known to the attacker at the same memory address as the original program such that this other program reads the encrypted execution state.

In the microprocessor of this embodiment, the encrypted execution state contains the program encryption key, and this key will be used in decrypting the encrypted program at a time of restart. Because of this mechanism, even when the other program other than the original program attempts to read the execution state, the key for does not match so that the program cannot be decrypted correctly and the program cannot be executed according to the intention of the attacker. Thus the second attacking method is impossible in the microprocessor of this embodiment.

This effect cannot be realized by simply encrypting the execution state itself by the public key of the processor, but can be realized by encrypting the decryption key of the original program and the execution state integrally.

Note that, in order to maximize this effect, values of the registers (R0 to R31) and the common key Kx should preferably be stored in the identical cipher block at a time of the encryption using the public key.

[Data Protection]

In the microprocessor of this embodiment, the encryption of the data is not accounted, but it should be apparent to those skilled in the art that it is possible to add the data encryption function to the microprocessor of this embodiment similarly as the data encryption in the microprocessor

for supporting the virtual memory which will be described in the second embodiment.

Referring now to FIG. 3 to FIG. 14, the second embodiment of a tamper resistant microprocessor according to the present invention will be described in detail.

In this embodiment, the microprocessor according to the present invention will be described for an exemplary case of using an architecture based on the widely used Pentium Pro microprocessor of the Intel corporation, but the present invention is not limited to this particular architecture. In the following description, features specific to the Pentium Pro microprocessor architecture will be noted and applications to the other architectures will be mentioned.

Note that the Pentium Pro architecture distinguishes three types of addresses in the address space including physical addresses, linear addresses and logical addresses, but the linear addresses in the Pentium terminology will also be referred to as logical addresses in this embodiment.

In the following description, the protection implies the protection of secrets of applications (that is the protection by encryption), unless otherwise stated. Consequently, the protection in this embodiment should be clearly distinguished from the ordinarily used concept of protection, that is the prevention of disturbances on the operations of the other applications due to the operation of some application. However, in the present invention, it is assumed that the operation protection mechanism in the ordinary sense is of course provided by the OS (although the description of this aspect will be omitted as it is unrelated to the present invention), in parallel to the protection of secrets of applications according to the present invention.

Also, in the following description, a machine language instructions that are executable by the processor will be referred to as instructions, and a plurality of instructions will be collectively referred to as an execution code or an instruction stream. A key used in encrypting the instruction stream will be referred to as the execution code encryption key.

Also, in the following description, the secret protection mechanism will be described as protecting secrets of applications under the management of the OS, but this mechanism can also be utilized as a mechanism for protecting the OS itself from alteration or analysis.

FIG. 3 shows a basic configuration of the microprocessor according to this embodiment, and FIG. 4 shows a detailed configuration of the microprocessor shown in FIG. 3.

The microprocessor **101** has a processor core **111**, an instruction TLB (Table Lookup Buffer) **121**, an exception processing unit **131**, a data TLB (Table Lookup Buffer) **141**, a secondary cache **152**. The processor core III includes a bus interface unit **112**, a code and data encryption/decryption processing unit **113**, a primary cache **114**, and an instruction execution unit **115**.

The instruction execution unit **115** further includes an instruction fetch/decode unit **214**, an instruction table **215**, an instruction execution switching unit **216**, and an instruction execution completing unit **217**.

The exception processing unit **131** further includes a register file **253**, a context information encryption/decryption unit **254**, an exception processing unit **255**, a secret protection violation detection unit **256**, and an execution code encryption key and signature verification unit **257**.

The instruction TLB **121** further includes a page table buffer **230**, an execution code decryption key table buffer **231**, and a key decryption unit **232**. The data TLB **141** further includes a protection table management unit **233**.

The microprocessor **101** has a key storage region **241** for storing a public key Kp and a secret key Ks which are unique to this microprocessor. Now, consider the case of purchasing a desired execution program A from some program vendor and executing it. The program vendor encrypts the program A by using a common execution code encryption key Kcode ($E_{Kcode}[A]$) before supplying the execution program A, and sends the common key Kcode used for encryption in a form encrypted by using the public key Kp of the microprocessor **101** ($E_{Kp}[Kcode]$) to the microprocessor **101**. The microprocessor **101** is a multi-task processor which processes not only this execution program A but also a plurality of different encrypted programs in a quasi parallel manner (that is by allowing interruptions). Also, the microprocessor **101** obviously executes not only the encrypted programs but also plaintext programs.

The microprocessor **101** reads out a plurality of programs encrypted by using different execution code encryption keys from a main memory **281** external of the microprocessor **101** through the bus interface unit (reading function) **112**. The execution code decryption unit **212** decrypts these plurality of read out programs by using respectively corresponding decryption keys, and the instruction execution unit **115** executes these plurality of decrypted programs.

In the case of interrupting the execution of some program, the context information encryption/decryption unit **254** of the exception processing unit **131** encrypts information indicating the execution state up to an interrupted point of the program to be interrupted and the code encryption key of this program by using the public key of the microprocessor **101**, and writes the encrypted information into the main memory **281** as the context information.

In the case of restarting the interrupted program, the execution code encryption key and signature verification unit **257** decrypts the encrypted context information by using the secret key of the microprocessor **101**, verifies whether the execution code encryption key contained in the decrypted context information (that is the execution code encryptionb key of the program scheduled to be restarted) coincides with the original execution code encryption key of the interrupted program, and restarts the execution of the program only when they coincide.

Here, before describing the detailed configuration and functions of the microprocessor **101**, the processing procedure for the execution of plaintext instructions and the execution of encrypted programs by the microprocessor **101** will be outlined.

When the microprocessor **101** executes a plaintext instruction, the instruction fetch/decode unit **214** attempts to read the content of an address indicated by a program counter (not shown) from an L1 instruction cache **213**. If the content of the specified address is cached, the instruction is read out from the L1 instruction cache **213**, sent to the instruction table **215**, and executed. The instruction table **215** is capable of executing a plurality of instructions in parallel, and requests reading of data necessary for carrying out the execution to the instruction execution switching unit **216** and receives the data. When the instructions are executed in parallel and their execution results are determined, the execution results are sent to the instruction execution completing unit **217**. The instruction execution completing unit **217** writes the execution result into the register file **253** when the operation target is a register inside the microprocessor **101**, or into an L1 data cache **218** when the operation target is a memory.

The content of the L1 data cache **218** is cached once again by an L2 cache **152** under the control of the bus interface unit **112**, and written into the main memory **281**. Here, the virtual memory mechanism is used, where a correspondence

between the logical memory address and the physical memory address is defined by a page table shown in FIG. **5**.

The page table is a data structure placed on the physical memory. The data TLB **141** actually carries out a conversion from the logical address to the physical address, and at the same time manages the data cache. The data TLB **141** reads a necessary portion of the table according to a top address of the table indicated by a register inside the microprocessor **101**, and carries out the operation for converting the logical address into the physical address. At this point, only the necessary portion of the page table is read out to a page table buffer **234** according to the logical address to be accessed, rather than reading out the entire page table on the memory to the data TLB **141**.

The basic cache operation is stable regardless of whether the instructions of the program are encrypted or not. Namely, a part of the page table is read out to the instruction TLB **121**, and the address conversion is carried out according to the definition contained therein. The bus interface unit **112** reads instructions from the main memory **281** or the L2 cache **152**, and instructions are stored in the L1 instruction cache **213**. The reading of instructions out to the L1 instruction cache **213** is carried out in units of a line formed by a plurality of words, which enables a faster access than the reading in word units.

The address conversion utilizing the same page table on the physical memory is also carried out for the processing target data of the executed instructions, and the execution of the conversion is carried out at the data TLB **141** as described above.

The operation up to this point is basically the same as the general cache memory operation.

Next, the operation in the case of executing an encrypted program will be described. In this embodiment, it is assumed that the execution codes for which secrets are to be protected are all encrypted, and the encrypted execution codes will also be referred to as protected codes. In addition, a range of the protection by the same encryption key will be referred to as a protection domain. Namely, a set of codes protected by the same encryption key is belonging to the same domain, and codes protected by different encryption keys are belonging to different protection domains.

First, the execution codes of a program encrypted by the secret key scheme block cipher algorithm are stored on the main memory **281**. A method for loading the encrypted program transmitted from a program vendor will be mentioned below.

A cipher block size of the execution codes can be any value as long as two to the power of the block size coincides with a line size that is a unit for reading/writing with respect to the cache memory. However, if the block size is so small that a block length coincides with an instruction length, there arises a possibility for analyzing the instruction easily by recording a correspondence between encrypted data and a predictable portion of the instruction such as a top portion of a sub-routine. For this reason, in this embodiment, the blocks are interleaved such that there is a mutual dependency among data in the blocks and the encrypted block contains information on a plurality of instruction words or operands. In this way, it is made difficult to set a correspondence between the instruction and the encrypted block.

FIGS. **7**A and **7**B show an example of the interleaving that can be used in this embodiment. In this example, it is assumed that the line size of the cache is 32 bytes and the block size is 64 bits (i.e., 8 bytes). As shown in FIG. **7**A, before the interleaving, one word is formed by 4 bytes, so that a word A is formed by 4 bytes of A**0** to A**3**. One line is formed by 8 words of A to H. When this is interleaved in units of 8 bytes corresponding to the block size of 64 bits, as shown in FIG. **7**B, A**0**, B**0**, . . . , H**0** are arranged in the first block corresponding to word **0** and word **1**, A**1**, B**1**, H**1** are arranged in the next block, and so on.

An attack can be made more difficult by setting a length of a region to be interleaved longer, but the interleaving of a region with a length longer than the line size makes the processing more complicated and lowers the processing speed because the decryption/encryption of one cache line would depend on reading/writing of another line. Thus it is preferable to set a range for interleaving within a range of the cache line size.

Here the method for interleaving data of blocks is used such that there is a mutual dependency among data in a plurality of blocks contained in the cache line, but it is also possible to use the other method for generating a dependency among data blocks, such as the CBC (Cipher Block Chaining) mode of the block cipher.

The decryption key Kcode (which will also be referred to as the encryption key hereafter even in the case of decryption because the encryption key and the decryption key are identical in the secret key algorithm) of the encrypted execution codes is determined according to the page table. FIG. **5** and FIG. **6** show a table structure of the conversion from the logical address to the physical address.

A logical address **301** of the program counter indicates some value, and a directory **302** and a table **303** constituting its upper bits specify a page entry **307**-*j*. The page entry **307**-*j* contains a key entry ID **307**-*j*-*k*, and a key entry **309**-*m* to be used for decryption of this page is determined in a key table **309** according to this ID. The physical address of the key table **309** is specified by a key table control register **308** inside the microprocessor.

In this configuration, the ID of the key entry is set in the page entry rather than setting the key information directly, such that the key information in a large size is shared among a plurality of pages so as to save a limited size of a memory region on the instruction TLB **121**.

In further detail, the page table and key table information is stored into the instruction TLB **121** as follows. Only portions necessary for the access to the memory is read out from the page tables **306**, **307** and **311** to the page table buffer **230**, and from the key table **309** to the execution code decryption key table buffer **231**.

In a state of being stored on the main memory, a reference counter of the key object **309**-*m* which is an element of the key table **309** indicates the number of page tables that refer to this key object. In a state where the key object is read out to the execution code decryption key table buffer **231**, this reference counter indicates the number of page tables that refer to this key object and that are read out to the page table buffer **230**. This reference counter will be used for judgement at a time of deleting any unnecessary key object from the execution code decryption key table buffer **231**.

One of the features of this embodiment is that the key table entry has a fixed length but a key length used in each table is made variable in order to be able to deal with a higher cryptoanalytic power, and specified at a key size region of the key table. It implies that the secret key Ks unique to the microprocessor **101** is fixed but the length of Kcode to be used for encryption and decryption of the program can be changed by the specification of the key entry. In order to specify a position of the variable length key, the key entry **309**-*m* has a field **309**-*m*-**4** pointing to the key entry, which indicates an address of the key object **310**.

In the key object region **310**, the execution code encryption key Kcode is stored in a form E_{Ks}[Kcode] encrypted by

the public key algorithm using the public key Kp of the microprocessor **101**. In order to encrypt data safely in the public key algorithm, a large redundancy is necessary, so that a length of the encrypted data becomes longer than a length of the original data. Here, lengths of Ks and Kp are set to be 1024 bits, a length of Kcode is set to be 64 bits, which is extended to 256 bits by padding, and E[Kcode] is encrypted in a length of 1024 bits and stored in the key object region **310**. When Kcode is so long that it cannot be stored in 1024 bits, it is divided into a plurality of blocks of 1024 bits size each and stored.

FIG. **8** summarizes the information flow in the code decryption. A program counter **501** indicates an address "Addr" on an encrypted code region **502** on a logical address space **502**. The logical address "Addr" is converted into the physical address "Addr" according to the page table **307** that is read out to the instruction TLB **121**. At the same time, the encrypted code decryption key E[Kcode] is taken out from the key table **309**, decrypted by using the secret key Ks provided in the CPU at a decryption function **506**, and stored into a current code decryption key memory unit **507**. The common key Kcode for the code encryption is encrypted by using the public key Kp of the microprocessor **101** by the program vendor, and supplied along with the program encrypted by using Kcode, so that the user who does not know the secret key Ks of the microprocessor **101** cannot know Kcode.

After the program execution codes are encrypted by using Kcode and shipped, the program vendor keeps and manages Kcode safely such that its secret will not be leaked to a third party.

An entire key table **511** and an entire page table **512** are placed in a physical memory **510**, and their addresses are specified by a key table register **508** and a CR3 register **509** respectively. From the contents of these entire tables, only necessary portions are cached into the instruction TLB **121** through the bus interface unit **112**.

Now, when a content **503** corresponding to the physical address "Addr'" as converted by the instruction TLB **121** is read out by the bus interface unit **112**, this page is encrypted so that it is decrypted at a code decryption function **212**. The reading is carried out in units of the cache line size, and after the decryption in block units, the inverse processing of the interleaving described above is carried out. The decrypted result is stored in the L1 instruction cache **213**, and executed as an instruction.

Here, the method for loading the encrypted program and the relocation of the encrypted program will be described. For the loading of a program into the memory, there is a method in which a program loader changes an address value contained in the execution codes of the program in order to deal with a change of an address for loading the program, but this method is not applicable to the encrypted program. However, the relocation of the encrypted program is possible by using a method of realizing the relocation without directly rewriting the execution codes by utilizing a table called Jump table or IAT (Import Address Table).

Further details of the loading procedure and the relocation for general programs can be found, for example, in L. W. Allen et al., "Program Loading in OSF/1, USENIX winter, 1991, and the loading method and the relocation for the encrypted program can be found in Japanese Patent Application No. 2000-35898 of the applicants.

It is possible to protect the execution codes placed on the memory external of the processor by the above described method for decrypting the encrypted execution codes of the program, reading them out to the cache memory inside the processor, and executing them.

However, the execution codes that are decrypted into plaintext can exist inside the processor. Even if it is impossible to read them out directly from outside the processor, there is a possibility for the plaintext program to be read out and analyzed by the other programs that are operated in the same processor.

In this embodiment, the key decryption processing by using the secret key **241** and the key decryption unit **232** of the instruction TLB **121** is not carried out at a time of data reading into an L1 data cache **218**. When the data reading is carried out with respect to an encrypted page for which an encryption flag **307**-J-E is set to "1" in the page table, either non-decrypted original data or data of a prescribed value "0" will be read out, or else an exception occurs such that the normally decrypted data cannot be read out. Note that when the encryption flag **307**-*j*-E in the page table is rewritten, the decrypted content of the corresponding instruction cache will be invalidated.

By this mechanism, it becomes impossible for the other programs (including the own program) to read the execution codes of the encrypted program as data, and decrypt them by utilizing functions of the processor.

Also, the other programs cannot explicitly read data in the instruction cache, so that the safety of the execution codes can be guaranteed. The safety of the data will be described below.

Because the encrypted execution codes can be executed in this way, in the microprocessor of this embodiment, by selecting the encryption algorithm and parameters appropriately, it can be made cryptographically impossible for a party who does not know the true value of the execution code encryption key Kcode to analyze the operation of the program by de-assembling the execution codes.

Thus the user cannot know the true value of the execution code encryption key Kcode, and it can be made cryptographically impossible for the user to make an alteration according to the user's intention such as illegal copying of the contents handled by the application by altering a part of the encrypted program.

Next, another feature of the microprocessor of this embodiment regarding the encryption, signature and its verification for the context at a time of interrupting the program execution under the multi-task environment will be described.

The execution of the program under the multi-task environment is often interrupted by the exception. Normally, when the execution is interrupted, a state in the processor is saved on the memory, and then the original state is recovered at a time of restarting the execution of that program later on. In this way, it becomes possible to execute a plurality of programs in a quasi parallel manner and accept the interruption processing. This information on the state at a time of the interruption is called the context information, the context information contains information on registers used by the application, and in some cases, information on registers that are not explicitly used by the application is also contained in addition.

In the conventional processor, when the interruption occurs during the execution of some program, the control is shifted to the execution codes of the OS while the register state of the application is maintained, so that the OS can check the register state of that program to guess what instructions were executed, or alter the context information maintained in a plaintext form during the interruption so as to change the operation of the program after the restart of the execution of that program.

In view of this fact, in this embodiment, when the interruption occurs during the execution of the protected

codes, the context of the execution immediately before that is encrypted and saved while all the application registers are either encrypted or initialized, and a signature made by the processor is attached to the context information. The signature is verified at a time of recovery from the interruption, to check whether the signature is proper or not. When the improper signature is detected, the recovery is stopped so that the illegal alteration of the context information by the user can be prevented. At this point, the encryption target registers are user registers **701** to **720** shown in FIG. **9**.

In the Pentium Pro architecture, there is a hardware mechanism for assisting the saving of the context information of the process into the memory and its recovery. A region for saving the state is called TSS (Task State Segment). In the following, an exemplary case of applying the present invention to this mechanism will be described, but the present invention is not limited to the Pentium Pro architecture, and equally applicable to any processor architectures in general.

The saving of the context information in conjunction with the exception occurrence takes place in the following case. When the exception occurs, an entry corresponding to the interruption cause is read out from a table called IDT (Interrupt Descriptive Table) for describing the exception processing, and the processing described there is executed, When the entry indicates a TSS, the context information saved in the indicated TSS is recovered to the processor. On the other hand, the context information of the process that has been executed up until then is saved in the TSS region specified by a task register **725** at that point.

Using this automatic context saving mechanism, it is possible to save the entire state of the application including the program counter and the stack pointer, and detect any alteration at a time of the recovery by verifying the signature. However, when this automatic context saving is used, apart from the fact that a large overhead will be caused by the context switching, there arises a problem that it is impossible to carry out the interruption processing without using the TSS.

In order to reduce the overhead due to the interruption processing, or to maintain the compatibility with the existing programs, it is preferable not to use the automatic context saving mechanism, but in such a case, the program counter will be saved on the stack and cannot be a target of the verification, so that it can be a target of the alteration by the malicious OS. These two cases should preferably used in their proper ways according to the purpose. For this reason, the microprocessor of this embodiment adopts the automatic context saving with respect to the protected (encrypted) execution codes as a result of attaching more importance to the safety. The registers to be automatically saved may not necessarily be all registers.

The context saving and recovery processing in this embodiment has the following three major features.

(1) The contents of the saved context can be decrypted only by the microprocessor that generated the context and a person who knows the encryption key Kcode of the program that generated the context.

(2) In the case where the program protected by some execution code encryption key X is interrupted and its context is saved, its restart processing cannot be applied to the restart of a non-protected program or a program encrypted by another execution code encryption key Y. Namely, the program to be recovered from the interruption cannot be replaced by another program at a time of the restart.

(3) The recovery of the altered context is prohibited. Namely, if the saved context is altered, that context will not be recovered.

By the above feature (1), it is possible to maintain the safety of the context information while enabling the analysis of the context information by the program vendor. The fact that the program vendor has a right to analyze the context information is important in order to maintain the quality of the program by analyzing causes of any trouble that occurred according to a condition by which the program is used by the user.

The above feature (2) is effective in preventing a situation where an attacker applies the context generated by the execution of a program A to another encrypted program B and restarts the program B from a known state saved in the context in order to analyze secrets of the data or the codes contained in the program B or alter the operation of the program B. This function is also a prerequisite for the data protection to be described below in which each one of a plurality of applications maintains own encrypted data exclusively and independently from the others.

By the above feature (3), it is possible to strictly eliminate the alteration of the context information utilizing an occasion of the restart of the program.

The reason for providing such a function is that simply encrypting the context information according to the secret information of the processor can protect the context information from the alteration according to the intention of the attacker, but it is impossible to eliminate a possibility for the random alteration of the context that results in the restart of the program from a state with random errors.

In the following, the context saving and verification method incorporating the above three features will be described in further detail.

<Context Saving Processing>

FIG. **10** shows the context saving format in this embodiment conceptually. It is assumed that the interruption due to the hardware or software related cause has occurred during the execution of the protected program. If the IDT entry corresponding to the interruption indicates a TSS, the execution state of the program up to that point is encrypted, and saved as the context information in a TSS indicated by the current task register **725** (rather than the indicated TSS itself). Then, the execution state saved in the TSS indicated by the IDT entry is recovered to the processor. If the IDT entry does not indicate a TSS, only the encryption or the initialization of the current registers is carried out, and the saving into the TSS does not takes place. Of course the restart of that program becomes impossible in that case. Note however that the system registers including a part of the flag registers and the task register are excluded from a target of the encryption or the initialization of the registers for the sake of continuation of the OS operation.

The contents of the context shown in FIG. **10** are actually interleaved, encrypted in block units and stored in the memory. Here the information items to be saved will be described first. At a top, stack pointers and user registers **802** to **825** corresponding to respective privileged modes are provided, and one word **826** indicating a TSS size and the presence/absence of the encryption is placed next. This indicates whether the TSS in which the processor is saved is encrypted or not. Even in the case where the TSS is encrypted, this region will be maintained in a plaintext form without being encrypted.

After that, data encryption control register (CY0 to CY3) regions **827** to **830** that are added for the purpose of the data protection are placed, and a padding **831** for adjusting the size to the block length is placed. Finally, a value $E_{Kcode}[Kr]$ **832** in which a key Kr used in encrypting the context is encrypted by the secret key algorithm using the execution

code encryption key Kcode, a value E_{Kp}[Kr] **833** in which the key Kr used in encrypting the context is encrypted by using the public key Kp of the processor, and a signature S_{Ks}[message] **834** using the secret key Ks of the processor with respect to them all are placed. Also, a region **801** for a link to the previous task that maintains a call up relationship among tasks is saved in a plaintext form in order to enable the task scheduling by the OS.

These execution code encryption and signature generation are carried out by the context information encryption/decryption unit **254** in the exception processing unit **131** shown in FIG. **4**, which is based on a function independent from the encryption of the processing target data of the execution codes. At a time of saving the context information in the TSS, even if some encryption is specified in an address of the TSS by the other data encryption function, this specification is ignored and the context information is saved in a state in which the context is encrypted. This is because the encryption attributes of the data encryption function are specific to each protected (encrypted) program so that the restart of some program cannot depend on that function.

In encrypting the context, a word in the TSS size region **826** to be recorded in a plaintext form is replaced to a value "0". Then, the interleaving similar to that explained with references to FIGS. **7**A and **7**B is applied, and the context is encrypted. At this point, the padding **831** is set to a size that enables the appropriate interleaving in accordance with the encryption block size.

Here, the reason for not encrypting the register values directly by the public key Kp of the processor or the execution code encryption key Kcode is to enable the analysis of the encrypted context by both the program vendor and the processor while prohibiting the decryption of the context by the user.

The program vendor knows the execution code encryption key Kcode so that the program vendor can obtain the encryption key Kr of the context by decrypting E_{Kcode}[Kr] **832** by using Kcode. Also, the microprocessor **101** can obtain the encryption key Kr of the context by decrypting E_{Kp}[Kr] **833** by using the own secret key Ks. Namely, the program vendor can analyze the trouble by decrypting the context information without knowing the secret key of the microprocessor of the user, and the microprocessor **101** itself can restart the execution by decrypting the context information by using the own secret key Ks. The user who does not have either key cannot decrypt the saved context information. Also, the user who does not know the secret key Ks of the microprocessor **101** cannot forge the context information and the signature S_{Ks}[message] with respect to E_{Kcode}[Kr] and E_{Kp}[Kr].

In order to enable the mutually independent decryption of the context information by the program vendor and the microprocessor, it is also possible to consider a method for encrypting the context information directly by using Kcode. However, in the case where the register state is already known, there is a possibility for the known-plaintext attack against the execution code encryption key Kcode. Namely, when a value of the key for encrypting data is fixed, the following problem arises. Consider the case of executing a program which reads a data input by the user and writes it into a working memory temporarily by encrypting it. The data that are to be encrypted and written into the working memory can be ascertained by monitoring the memory, so that the user can repeat the input many times by changing the input value and obtain the corresponding encrypted data. This implies that the chosen-plaintext attack of the cryptoanalysis theory is possible.

The known-plaintext attack is not fatal to the secret key algorithm, but it is still preferable to avoid that. For this reason, a random number Kr is generated at a random number generation mechanism **252** of the exception processing unit **131** at each occasion of the context saving, and supplied to the context information encryption/decryption unit **254**. The context information encryption/decryption unit **254** encrypts the context by the secret key algorithm using the random number Kr. Then, the value E_{Kcode}[Kr] **832** in which the random number Kr is encrypted by the same secret key algorithm using the execution code encryption key Kcode is attached. The value E_{Kp}[Kr] **833** is obtained by encrypting the random number Kr by the public key algorithm using the public key Kp of the microprocessor.

Here, the random number is generated by the random number generation mechanism **252**. In the case where the program is encrypted, normally there is no change in the program codes so that the corresponding plaintext codes cannot be acquired illegally as long as the operation is not analyzed. In this case, there is a need to carry out the "ciphertext-only attack" in order to cryptoanalyze, so that it is very difficult to find the encryption key. However, in the case where the data entered by the user are to be stored into the memory by encrypting them, the user can freely select the input data. For this reason, it is possible for the user to make the "chosen-plaintext attack" against the encryption key which is far more effective than the "ciphertext-only attack".

Against the chosen-plaintext attack, it is possible to adopt a measure for enlarging the search space by adding a random number called "salt" into the plaintext to be protected. However, it is very tedious to implement the saving into the memory in a form where the "salt" random number is incorporated in every data at the application programming level, so that this can cause the lowering of the programming efficiency and performance.

For this reason, the random number generation mechanism **252** generates the random number (encryption key) for encrypting the context at each occasion of the context saving. As the encryption key can be selected arbitrarily, there is also an effect that the safe communications between processes or between processes and devices can be realized faster. This is because the speed for encrypting data by the hardware at a time of the memory access is far slower in general than the speed for encrypting data by the software.

On the contrary, if the value of the encryption key for the data region is limited to a prescribed value such as that identical to the execution code encryption key for example, then it becomes impossible to use the data encryption function of the processor for the other programs encrypted by the other encryption keys or the sharing of the encrypted data with the devices, so that it becomes impossible to take advantage of the fast hardware encryption function provided in the processor.

Note that the decryption of the encrypted random number E_{Kcode}[Kr] **832** that takes place at a time of the restart and the generation of the signature **834** can be based on any algorithm and secret information as long as a condition that they can be carried out only by the microprocessor **101** is satisfied. In the above example, the secret key Ks unique to the microprocessor **101** (which is also used for the decryption of the execution code encryption key Kcode) is used for both, but respectively different values may be used for these purposes.

Also, the saved context contains a flag indicating the presence/absence of the encryption, so that the encrypted

context information and the non-encrypted context information can coexist according to the need. The TSS size and the flag indicating the presence/absence of the encryption are stored in a plaintext form so that it is easy to maintain the compatibility with respect to the past programs.

<Processing for Restarting the Interrupted Program>

At a time of restarting the process by recovering the context, the OS issues a Jump or call instruction with respect to a TSS descriptor indicating the saved TSS.

Returning now to FIG. **4**, the execution code encryption key and signature verification unit **257** if the exception processing unit **131** verifies the signature S_{Ks}[message] **834** by using the secret key Ks of the processor first, and sends the verification result to the exception processing unit **255**. In the case where the verification result is failure, the exception processing unit **255** stops the restart of the execution of the program, and causes the exception. By this verification, it is possible to confirm that the context information is surely generated by the proper microprocessor **101** that has the secret key and not altered.

When the verification of the signature succeeds, the context information encryption/decryption unit **254** obtains the random number Kr by decrypting the context encryption key E_{Kp}[Kr] **833** by using the secret key Ks. On the other hand, the execution code encryption key Kcode corresponding to the program counter (EIP) **809** is taken out from the page table buffer **230**, and sent to the current code encryption key memory unit **251**. The context information encryption/decryption unit **254** decrypts E_{Kcode}[Kr] by using the execution code decryption key Kcode, and sends the result to the execution code encryption key and signature verification unit **257**. The execution code encryption key and signature verification unit **257** verifies whether the decryption result of E_{Kcode}[Kr] **832** coincides with the decryption result of the microprocessor using the secret key Ks or not. By this verification, it is possible to confirm that this context information is generated by the execution of the execution codes encrypted by using the secret key Kcode.

If this verification of the execution code encryption key with respect to the context information is not carried out, it would become possible for the user to make an attack by producing codes encrypted by using any suitable secret key Ka and applies the context information obtained by executing these codes to the codes encrypted by the other secret key Kb. The above verification eliminates a possibility of this attack and guarantees the safety of the context information for the protected codes.

This object can also be achieved by adding a secret execution code encryption key Kcode to the context information, but in this embodiment, by the use of the value E_{Kcode}[Kr] in which a secret random number Kr used in encrypting the context information is encrypted by using the execution code encryption key Kcode selected by the program vendor, it is possible to reduce the amount of memory required for saving the context information so as to achieve the effects of the fast context switching and the memory saving. This also enables the feedback of the context information to the program creator.

Now, when the verification of the execution code encryption key and the verification of the signature by the execution code encryption key and signature verification unit **257** both succeed, the context is recovered to the register file **253**, and the program counter value is also recovered so that the control is returned to an address at a time of the execution interruption that caused to generate this context.

When either one of these verifications fails so that the exception processing unit **255** causes the exception to occur, an exception occurrence address indicates an address at which the jump or call instruction is issued. Also, a value indicating illegality of the TSS is stored into an interruption cause field in the IDT table, and an address of a jump target TSS is stored into a register that stores an address that is the cause of the interruption. In this way, the OS can learn the cause of the context switching failure.

Note that, in order to realize the faster restart processing, it is also possible to use a configuration in which the supply of the execution state encrypted by the context information encryption/decryption unit **254** to the register file **253** and the verification processing by the execution code encryption key and signature verification unit **257** are carried out in parallel, and the subsequent processing is stopped when the verification fails.

The safety of this encryption scheme using a random number depends on the impossibility to predict a random number sequence used, and a method for generating by hardware a random number that is very hard to predict is disclosed in Onodera, et al., Japanese Patent No. 2980576.

The analysis of the context information by the program vendor is important in improving the quality of the program by analyzing the causes of any trouble in the program that occurred according to a condition by which the program is used by the user. In this embodiment, in view of this fact, the above described scheme for realizing both the safety of the context and the capability of the context information analysis by the program vendor is employed, but it is also true that the use of this scheme increases the overhead of the context saving.

Moreover, the verification of the context information by using the signature made by the microprocessor prevents the execution of the protected codes in the illegal context information by using a combination of arbitrarily selected value and encryption key, but this additional protection also increases the overhead.

Consequently, in the case where there is no need for the capability of the context information analysis by the program vendor or a mechanism for eliminating the program restart using the Illegal context information, the context information containing information for identifying the execution code encryption key may be directly encrypted by using the secret key of the processor. Even in such a case, it is still possible to make the intentional alteration of the context cryptographically impossible, and prevent the context information from being applied to a program encrypted by using a different encryption key.

Here, the context saving format will be described further. Its relationship with the operation will be described later.

In FIG. **10**, an "R" bit **825-1** is a bit indicating whether the context is restartable or not. When this bit is set to "1", the execution can be restarted by recovering the state saved in the context by the above described recovery procedure, whereas when this bit is set to "0", the restart cannot be made. This has an effect of preventing the restart of the context in which the illegality is detected during the execution of the encrypted program so as to limit the restartable contexts to only those in the proper states.

A "U" bit **825-2** is a flag indicating whether the TSS is a user TSS or a system TSS. When this bit is set to "0", the saved TSS is the system TSS, and when this bit is set to "1", the saved TSS is the user TSS. The TSS that will be saved and recovered through the task switching accompanied by the change of the privilege from the exception entry as described above or through a task gate call up is the system TSS.

The difference between the system TSS and the user TSS lies in whether a task register indicating a TSS saving

location of the currently executed program is to be updated or not at a time of the recovery of the TSS. In the recovery of the system TSS, the task register of the currently executed program will be saved in the link to the previous task region **801** of the TSS to be newly recovered, and the segment selector of the new TSS will be read into the task register. On the other hand, in the recovery of the user TSS, the update of the task register value will not be carried out. The user TSS is aimed only at the saving and the recovery of the register state of the program so that it is not accompanied by the change of the privileged mode.

The exception includes a software interrupt used for the system call up from the application program. In the case of the software interrupt for the purpose of the system call up, the general purpose register is often used for the parameter exchange, and there can be cases where the context information encryption can obstruct the parameter exchange.

The software interrupt is generated by the application itself, so that it is possible for the application to destroy information of the registers that have secrets, prior to the generation of the software interrupt. Under the presumption of such conditions, it is possible to use a scheme in which the encryption of the registers is not carried out only in the case of the software interrupt. Of course, in such a case, the application program creator should take this fact into consideration and design the program such that the secrets of the program can be protected.

Next, the suppression of the plaintext program debugging function will be described.

The processor has a step execution function which causes the interruption whenever one instruction is executed, and a debugging function which causes the exception whenever a memory access with respect to a specific address is made. These functions may be useful for the development of programs but they can impair the safety of programs that are encrypted for the purpose of the secret protection. Consequently, in the microprocessor of this embodiment, such debugging functions are suppressed during the execution of the encrypted program.

The instruction TLB **121** can Judge whether the currently executed code is protected or not (encrypted or not). During the execution of the protected code, two debugging functions including a debug register function and a step execution function are prohibited in order to prevent an intrusion of the encrypted program analysis from a debug flag or a debug register.

The debug register function is a function in which a memory access range and an access type such as reading/writing as the execution code or data are set in advance into a debug register provided in the processor such that the interruption is caused whenever a corresponding memory access occurs. In this embodiment, during the execution of the protected code, the contents set in the debug register will be ignored so that the interruption for the purpose of the debugging will not occur. Note however that the case where a debug bit is set in the page table is excluded from this rule. The debug bit in the page table will be described later.

During the execution of a non-protected (plaintext) code, the interruption will be caused whenever one instruction is executed if a step execution bit in an EFLAGS register of the processor is set, but during the execution of the protected code, this bit will also be ignored so that the interruption will not occur.

In this embodiment, in addition to the encryption of the execution codes for the purpose of preventing the analysis, these functions make the analysis of the program by the user difficult by preventing the dynamic analysis of the program using the debug register or the debug flag.

<Data Protection>

Next, the protection of the processing target data of the execution codes will be described.

In this embodiment, the encryption attributes for protecting data are defined in four registers CY0 to CY3 that are provided inside the microprocessor **101**. They correspond to regions **717** to **720** shown in FIG. **9**. In FIG. **9**, details of the registers CY**0** to CY**2** are omitted, and only details of the register CY**3** are shown.

Elements of the encryption attribute will now be described by taking the CY**3** register **717** as an example. Upper bits of the logical address indicating a top of the region to be encrypted are specified in a base address **717-1**. The size of the region is specified in a size region **717-4**. A size is specified in units of the cache line so that there is an invalid portion at the lower bits. A data encryption key is specified in a region **717-5**. Here the secret key algorithm is used so that the region **717-5** is also used for the decryption key, When a value of the encryption key is specified as "0", it implies that the region indicated by that register is not encrypted.

Among the specifications of the regions, CY**0** is given the highest priority, and CY**1** to CY**3** are given sequentially lower priorities in this order. For example, When the regions specified by CY**0** and CY**1** overlap, the attributes of CY**0** are given the priority over those of CY**1** in that region. Also, the definition of the page table is given the highest priority in the case of a memory access as the execution code rather than as the processing target data.

A debug bit **717-4** is used in selecting whether the data operation in the debugging state is to be carried out in an encrypted state or in a plaintext state. Details of the debug bit will be described later.

FIG. **12** shows the information flow in the encryption/decryption of the processing target data of the execution codes. Here, the data protection is made only in the state where the code is protected, that is the code is executed in an encrypted state. Note however that the case where the code is executed in the debugging state to be described below will be excluded from this rule. When the code is protected, the contents of the data encryption control registers (which will be also referred to as the encryption attribute registers or the data protection attribute registers) CY**0** to CY**3** are read from the register file **253** shown in FIG. **4** to a data encryption key table **236** provided inside the data TLB **141**.

When some instruction writes data into a logical address "Addr", the data TLB **141** Judges whether the logical address "Addr" is contained in ranges of CY**0** to CY**3** or not by checking the data encryption key table **236** (see FIG. **4**). As a result of the Judgement, if the encryption attribute is specified, the data TLB **141** commands the code encryption function **212** to encrypt the memory content by the specified encryption key at a time of the memory writing of a corresponding cache line from the L1 data cache **218** to the memory.

Similarly, in the case of reading, if the target address has the encryption attribute, the data TLB **141** commands the data decryption function **219** to decrypt the data by the specified encryption key at a time of the reading of a cache line out to the corresponding L1 data cache **218**.

In this embodiment, the data encryption attributes are protected from the illegal rewriting including the privilege of the OS by placing all the data encryption attributes for the data encryption in the registers inside the microprocessor **101** and saving the contents of the registers at a time of the execution interruption as the context information in a safe

form into a memory (the main memory **281** of FIG. **4**, for example) external of the microprocessor **101**.

The data encryption/decryption is carried out in units of the cache line that is interleaved as described above in relation to the context encryption. For this reason, even when one bit of the data on the L1 cache **114** is rewritten, the other bits in the cache line will be rewritten on the memory. The execution of the data reading/writing is carried out collectively in units of the cache line, so that the increase of the overhead is not so large, but it should be noted that the reading/writing with respect to the encrypted memory regions cannot be carried out in units less than or equal to the cache line size.

In the above, the method for protecting the data by encryption in this embodiment has been described. By this method, on the main memory, it is possible to process the encrypted data by encrypting them inside the processor by using the encryption key and the memory range specified by the application program, and read/write them as plaintext data from a viewpoint of the application.

Next, two mechanisms for preventing reading of the data stored in a plaintext form in the cache memory inside the processor by a program other than the encrypted programs that has read these data (which will be referred to as the other program) will be described.

First, the program is identified by its encryption key. This identification is made by using a key object identifier used at a time of decrypting the currently executed instruction inside the processor. Here, a value of the key itself may be used for this identification, but a value of the execution code decryption key has a rather large size of 1024 bits before the decryption or of 128 bits after the decryption which would require an increase of the hardware size, so that the key object identifier which has a total length of only 10 bits is used.

The L1 instruction cache **213** in which the decrypted execution codes are to be stored has an attribute memories in correspondences to the cache lines. When the decrypted execution codes are stored into the L1 instruction cache **213** by the code decryption function **212**, the key object identifier is written into the attribute memory.

Also, in the case of reading the encrypted data from the memory and decrypting it, the contents of the data protection attribute registers **CY0** to **CY3** are read out from the register file **253** to a protection table management function **233** of the data TLB **141**. At this point, the key object identifier corresponding to the currently executed instruction is also read from the current code encryption key memory unit **251** at the same time and maintained in the protection table management function **233**.

Similarly as in the case of the instruction cache, the data cache **218** has attribute memories in correspondence to the cache lines. When the data read out from the memory is decrypted by the data decryption function **219** and stored into the L1 data cache **218**, the key object identifier is written into the attribute memory from the protection table management function **233**.

When some instruction is executed and the data referring is carried out, the key object identifier written in the attribute of the data cache and the key object of that instruction in the instruction cache are compared by the secret protection violation detection unit **256**. If they do not coincide, the exception of the secret protection violation occurs and the data referring fails. In the case where the attribute of the data cache indicates a plaintext, the data referring always succeeds.

Note that, when the attributes of the instruction and the data do not coincide, instead of causing the exception, it is also possible to discard the content of this data cache and re-read the data from the memory once again.

For example, consider program-1 and program-2 for which the execution code encryption key as well as the data protection attribute registers **CY0** to **CY3** are different. If the encrypted data referred and written into the cache by the program-1 is to be referred by the program-2, the program-2 will read out a different data. This operation is in accord with the purpose of protecting secrets.

If two programs have the same data encryption key and data at the same address are referred by them, the same data will be read so that this data can be shared between them.

In this way, in this embodiment, data generated by some program-1 can be protected from being referred by another program-2 by providing a function for maintaining attributes of the instruction to be executed and the data indicating programs to which they originally belong, and comparing the attributes to see if they coincide or not at a time of the data referring due to the instruction execution.

<Entry Gate>

In this embodiment, the cases where the control can be shifted from the non-protected code to the protected code are limited only to the following two cases:

(1) the case where the context encrypted by using the execution code encryption key (that is, the context having a random number) that coincides with a restart address is to be restarted; and

(2) the case where the control is shifted from a non-protected code to an entry gate instruction ("egate" instruction) of the protected code. by the execution of the consecutive codes or by a Jump or call instruction.

This limitation is placed in order to prevent an attacker from obtaining information on code fragments by executing the code from arbitrary position. The procedure for the above (1) has already been described in relation to the context recovery. Namely, the control is shifted to the execution of the protected code only when it is verified that the context information matching with the execution code encryption key of the code that was executed immediately before the interruption is contained, and that the proper signature given by the microprocessor **101** is attached.

The above (2) is a processing for prohibiting a transition to the execution of the protected code unless a special instruction called entry gate ("egate") instruction is executed at the beginning of the control in the case of shifting the control from the non-protected code to the protected code.

FIG. **11** shows a procedure for switching a protection domain based on the entry gate instruction. The microprocessor **101** is maintaining the encryption key of the currently executed code in the current code encryption key memory unit **251** (see FIG. **4**) of the exception processing unit **131**. First, whether the value of this key is changed in conjunction with the execution of the instruction or not is judged (step **601**). When the change of the key value is detected (step **601** NO), whether the instruction executed in conjunction with the change is an entry gate ("egate") instruction or not is checked next (step **S602**). If it is the entry gate instruction, it implies that it is a proper instruction so that the control can be shifted to the changed code. Consequently, when it is Judged as an entry gate instruction (step **602** YES), this instruction is executed.

On the other hand, when it is Judged as not an entry gate instruction (step **602** NO), it implies that the interrupted instruction is an improper instruction. In this case, whether the instruction that was executed immediately previously is an encrypted (protected) instruction or not is judged (step **603**). If it is a non-protected instruction, the exception

processing can take place directly, but if it is a protected instruction, there is a need to carry out the exception processing while protecting that instruction.

Consequently, when it is judged as a non-protected instruction (step 603 NO), the exception processing is carried out directly, whereas when it is Judged as a protected instruction (step 6003 YES), the non-restartable exception processing is carried out while maintaining the protected state.

By this limitation of the control shifting, the direct shifting of the control from a plaintext code to a code at a location other than that of the entry gate instruction is prohibited. The context recovery implies the recovery of the state that was already executed once by that program through the entry gate. Consequently, the execution of the protected program must pass through the entry gate. By suppressing locations for placing the entry gate to the minimum necessary number in the program, there is an effect of preventing an attack for guessing a program structure by executing the program from various addresses.

Also, at this entry gate, the initialization of the data protection attribute registers is carried out. When the entry gate is executed, a random number Kr is loaded into a key region (a region 717-5 in CY3) of the data protection attribute registers CY0 to CY3 717 to 720 shown in FIG. 9 The encryption target top address is set to "0", the size is set to an upper limit of the memory, and the entire logical address space is set as the encryption target. If the debug attribute is not set in the execution code, the debug bit (717-3 in CY3) is set as non-debugging.

In other words, at a timing of the encryption code execution start, all the memory accesses are encrypted by using the random number Kr determined at a time of the entry gate execution. Also, in the execution code encryption control, the definition in the page table is given a higher priority as already mentioned above. This random number Kr is generated independently from the random number used in the context encryption.

By this mechanism, a protected program to be newly executed is set to be always encrypted by using a key determined randomly at a time of the start of all the memory accesses.

Of course, in this state the entire memory region is encrypted so that it is impossible to give parameters of the system call through the memory or exchange data with the other programs. For this reason, the program carries out the processing by sequentially adjusting its own processing environment by setting the data protection attribute registers such that the necessary memory region can be converted into plaintext so that it becomes accessible. By leaving the register CY3 with a lowest priority in the initial setting of being encrypted by using the random number, while setting the encryption key "0" as the plaintext access setting for the other registers, it is possible to reduce a risk of accessing an unnecessary region as a plaintext and writing data to be kept in secret by encryption out to a plaintext region by error.

The contents of the registers other than the data protection attribute registers are not encrypted even in the initialization at the entry gate, and pointers for specifying locations of stacks or parameters can be stored therein. However, cares should be taken in the processing of the program to be executed through the entry gate so that secrets of the program will not be stolen by calling up the entry gate by setting illegal values into the registers.

It is also possible to use a configuration for initializing all the registers other than the flags and the program counter, including the general purpose registers other than the data protection attribute registers, at the entry gate in the case of attaching more importance to the safety, even though this provision makes the programming more restricted and the efficiency poorer. Even in this case, the parameters such as stacks can be exchanged through a memory region specified by a relative address or an absolute address of the program counter. Note however that, similarly as in the case of the context saving, the system registers including a part of the flag registers and the task register are excluded from a target of the encryption or the initialization of the registers for the sake of continuation of the OS operation.

In this way, in the microprocessor 101 of this embodiment, the fragmental execution of the protected code, especially the illegal setting of the data protection state, is prevented, as the first instruction to be executed at a time of shifting the control from the program in the plaintext state to the protected program is limited to the entry gate instruction and the registers including the data protection attribute registers are initialized by the execution of the entry gate instruction.

Next, the execution control of the protected program will be described. First, the call up and the branching that are closed within the protection domain will be described, The call up within the protection domain is exactly the same as that for the usual programs. FIG. 13 shows the call up and the branching within the protection domain conceptually.

The execution of the code 1101 in the protection domain is started as a thread 1121 outside the protection domain is branched into an "egate" (entry gate) instruction of the protection domain. By the execution of the "egate" instruction, all the registers are initialized, and then the data protection attributes are set up sequentially by the execution of the program. The control is shifted to a branch target "xxx" 1111 in the protection domain by a "jmp xxx" instruction (processing 1122), and a "call yyy" instruction located at an address "ppp" 1112 is executed (processing 1123). The calling source address "ppp" 1112 is pushed into a stack memory 1102, and the control is shifted to a call target "yyy" 1113. When the processing at the call target is completed and a "ret" instruction is executed, the control is shifted to a return address "ppp" 1112 in the stack. There is no limitation on the execution control while the execution code encryption key remains the same.

Next, the call up and the branching from a protection domain to a non-protection domain will be described. For this control shifting, the execution of a special instruction and the operation of the user TSS to be described below will be carried out in order to avoid a shifting from a protection domain to a non-protection domain that is not intended by the program creator and to protect the data protection state.

FIG. 14 shows the call up and the branching from a protection domain to a non-protected domain conceptually, where an execution code 1201 of the protection domain and an execution code 1202 of the non-protection domain are placed in respective domains. Also, a user TSS region 1203 and a region 1204 for exchanging parameters with the non-protection domain are provided.

The execution begins when a thread 1221 executes the egate" instruction. The program of the protection domain saves the address of the user TSS region 1203 in a prescribed parameter region 1204 before calling up the code of the non-protection domain. Then, the code of the non-protection domain is called up by executing the "ecall" instruction. The "ecall" instruction takes two operands. One is a call target address, and the other is a saving target of the execution state. The "ecall" instruction saves the register state at a time of the call up (or more accurately the register state when the

program counter is in a state after the "ecall" instruction is issued) into a region specified by the operand "uTSS", in a format similar to that in the case of the encrypted TSS described above. In the following, this region will be referred to as a user TSS.

The difference between the user TSS and the system TSS lies in that, in the user register shown in FIG. 10, a U flag is set in a region 825-2 on the TSS. The difference in the operation will be described later. In the saving of the user TSS into the memory, the data protection attributes defined in the data protection attribute registers CY0 to CY3 by the user are not applied, similarly as in the case of the saving of the context information into the system TSS.

The call target code of the non-protection domain cannot exchange parameters because the registers are initialized by the execution of the "ecall" instruction. For this reason, the parameters are acquired from a prescribed address "param" 1204, and the necessary processing is carried out. There is no limitation on the programming in the non-protection domain. In the example of FIG. 14, a sub-routine "qqq" 1213 is called up (processing 1225). The call up from the protection domain can be adapted to the call up semantics of the sub-routine "qqq" by placing an adaptor code for copying stack pointer setting and the parameters to the stack, between "exx" and the call up of "qqq", for example. The processing result is sent to the calling source through the parameter region 1204 on the memory (processing 1226). When the processing of the sub-routine is completed, a "sret" instruction is issued in order to return the control to the calling source protection domain (processing 1227).

The "sret" instruction takes one operand for specifying the user TSS, unlike the "ret" instruction that has no operand. Here, the user TSS 1203 is specified indirectly as the recovery information through a pointer stored in the parameter region "param" 1204. The recovery of the user TSS by the "sret" instruction largely differs from the recovery of the system TSS in that the task register is not affected at all even when the user TSS is recovered. The task link field of the user TSS will be ignored. The recovery will fail when the system TSS with the U flag 825-2 set to "0" is specified in the operand of the "sret" instruction.

At a time of the execution of the recovery, the decryption of the execution state and the verification of the execution code encryption key and the signature already described above are carried out, and when the violation is detected, the exception of the secret protection violation will occur. When the verification succeeds, the execution is restarted from an instruction next to the calling source "ecall" instruction. This address is encrypted and signed in the user TSS, so that it is cryptographically impossible to forge this address. All the registers except for the program counter will be set back to the state before the call up, so that the code of the protection domain acquires the execution result of the sub-routine "exx" from the parameter region 1204.

At a time of shifting the control to the non-protection domain after the processing of the protection domain is completed, an "cjmp" instruction is used. The "cjmp" instruction does not carry out the saving of the state, unlike the "ecall" instruction. If the control is shifted from the protection domain to the non-protection domain by the instruction other than "ccall" and "cjmp", such as "Jmp" or "call", the exception of the secret protection violation occurs and the encrypted context information is saved in the TSS region (a region indicated by the task register) of the system. Note that the context information will be marked as non-restartable at this point. Note also that specifying an address in the protection domain as a jumping target of the "cjmp" instruction does not cause the violation.

This completes the description of a procedure for call up from the protection domain to the non-protection domain and newly added instructions used in that procedure.

At a time of the recovery of the user TSS by the application, an attack for substituting the user TSS by the OS which has privileges is not entirely impossible. However, the interchangeable TSS information in such a case is only the context information whose execution is always started through the "egate" and which is saved by the saving of the execution state caused by the interruption or by the user explicitly, as long as the execution code encryption key of the protection domain is managed correctly. A possibility for the leakage of the secrets of the application due to the interchange of this context information is quite small, and it is quite difficult for an attacker to guess what kind of the context information interchange is necessary in acquiring the secrets of the application.

The procedure for call up from the protection domain to the non-protection domain described above is also applicable to a procedure for shifting the control between the protection domains, if the instruction to be executed first at the call target is the "egate" instruction of the calling source side.

In this case, the call up between the protection domains can be carried out safely by encrypting the region for exchanging parameters between these protection domains, by using an encryption key that is shared by carrying out the authentication key exchange between these protection domains in advance.

As described, according to the microprocessor of the present invention, it becomes possible to prevent the illegal analysis by the OS or a third party by protecting both the execution codes and the processing target data of the execution codes by using the encryption, under the multi-task environment.

Also, it becomes possible to prevent the illegal rewriting of the encryption attributes in the case of saving the encrypted data.

Also, it becomes possible to protect the encrypted data from illegal attacks by using arbitrary random number Kr rather than a fixed key as the encryption key for the processing target data.

Also, it becomes possible to carry out the debugging in the plaintext state, and when errors are found, a feedback on the errors can be provided to the program vendor who knows the execution code encryption key.

Also, it becomes possible to prevent an increase of the memories in the microprocessor and suppress the cost of the microprocessor by saving information that required the secret protection such as the encryption attribute information on an external memory by attaching a signature of the microprocessor, reading only the necessary portion into the registers inside the microprocessor, and carrying out the verification of the signature at a time of reading. In this scheme, the safety against the substitution at a time of the reading can also be guaranteed.

It is also to be noted that, besides those already mentioned above, many modifications and variations of the above embodiments may be made without departing from the novel and advantageous features of the present invention. Accordingly, all such modifications and variations are intended to be included within the scope of the appended claims.

What is claimed is:

1. A microprocessor having a unique secret key and a unique public key corresponding to the unique secret key that cannot be read out to external, comprising:

a reading unit configured to read out a plurality of programs encrypted by using different execution code encryption keys from an external memory;

a decryption unit configured to decrypt the plurality of programs read out by the reading unit by using respective decryption keys;

an execution unit configured to execute the plurality of programs decrypted by the decryption unit;

a context information saving unit configured to save a context information for one program whose execution is to be interrupted, into the external memory or a context information memory provided inside the microprocessor, the context information containing information indicating an execution state of the one program and the execution code encryption key of the one program; and

a restart unit configured to restart an execution of the one program by reading out the context information from the external memory or the context information memory, and recovering the execution state of the one program from the context information;

wherein the context information saving unit is configured to generate a random number as a temporary key, to encrypt the context information, and to save an encrypted context information into the external memory, the encrypted context information containing a first value obtained by encrypting information indicating the execution state of the one program by using the temporary key and a second value obtained by encrypting the temporary key by using the public key;

the restart unit is configured to restart the execution of the one program by reading out the encrypted context information from the external memory decrypting the temporary key from the second value contained in the encrypted context information by using the secret key, decrypting the information indicating the execution state from the first value contained in the encrypted context information by using a decrypted temporary key, and recovering the execution state of the one program from a decrypted context information; and

the context information saving unit saves the encrypted context information that also contains a third value obtained by encrypting the temporary key by using the execution code encryption key of the one program.

2. The microprocessor of claim 1, wherein the restart unit decrypts a first temporary key from the second value contained in the encrypted context information by using the secret key and decrypts the information indicating the execution state from the first value contained in the encrypted context information by using the first decrypted temporary key, while decrypting a second temporary key from the third value contained in the encrypted context information by using the execution code encryption key of the one program, and restarts the execution of the one program only when the first decrypted temporary key coincides with the second decrypted temporary key.

* * * * *